TEKNILLINEN KORKEAKOULU
Teollisuuden tietotekniikan laboratorio

OHJELMOINTIVIRHEET YLEISIMPIEN TIETOTURVAONGELMIEN TAKANA

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 20.9.2004.

TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto
Teollisuuden tietotekniikka
Syksy 2004
Janne Parkkonen

Työn valvoja: Professori Juha Tuominen
Työn ohjaaja: Professori Juha Tuominen

TEKNILLINEN KORKEAKOULU	DIPLOMITYÖN TIIVISTELMÄ
Tietotekniikan osasto	

Tekijä	Päiväys:
Parkkonen, Janne	20.9.2004
	Sivumäärä:
	69

Työn nimi
Ohjelmointivirheet yleisimpien tietoturvaongelmien takana

Professuuri	Koodi
Teollisuuden tietotekniikka	T-126

Työn valvoja
Professori Tuominen, Juha

Työn ohjaaja
Professori Tuominen, Juha

Tietoturvaongelmat ovat nousseet monien vaarallisten matojen ja virusten johdosta tietotekniikan erääksi keskeisimmäksi haasteeksi. Samaan aikaan, kun sovellukset kasvavat ja ihmiset ovat yhä enemmän riippuvaisia tietotekniikasta, tietoturvaongelmat ja niiden kustannukset ovat kasvussa. Kuitenkin ohjelmointivirheet, jotka aiheuttavat suurimman osan tietoturvaongelmista, ovat suurimmaksi osaksi huonon syötteen tarkistuksen aiheuttamia ja suurin osa niistä voitaisiin korjata pienillä lisätarkistuksilla.

Työssä käydään ensin läpi tietoturvaan kuuluvia elementtejä sekä käsitellään tietoturvaa sovelluskehityksen kannalta. Tämän jälkeen työssä esitellään suunnittelumenetelmä, jonka tavoitteena on mahdollisimman tietoturvalliset sovellukset. Suunnittelumenetelmän jälkeen työssä esitellään yleisimmät tietoturvaongelmat siten, että ensin esitellään ongelman tausta minkä jälkeen näytetään esimerkki kyseisestä tietoturvaongelmasta. Esimerkeissä esitetään myös yksinkertainen hyökkäys, jolla pyritään havainnollistamaan tietoturvaongelman vaarallisuus.

Avainsanat
Ohjelmointivirheet, tietoturva

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF MASTER'S THESIS**
Department of Computer Science and Engineering

Author	Date:
Parkkonen, Janne	20.9.2004
	Pages: 69

Title of thesis

Programming errors behind the common security problems

Professorship	Professorship Code
Industrial Information Technology	T-126

Supervisor

Professor Tuominen, Juha

Instructor

Professor Tuominen, Juha

Largely spread viruses and worms have raised the information security related problems as one of the biggest challenges in information technology. As application sizes are growing and people are getting more dependant of computers, security problems and their costs are increasing. However, programming errors causing most of the problems are mostly due to bad input checking. Those errors could be fixed with little extra work.

This thesis starts with information security elements moving on to security of the software development. Then a design method for developing secure software is presented. Also the most common programming errors are addressed by using examples. Errors are described and explained and then presented with examples containing simple attack against a programming error. The purpose of the example is to present the danger inflicted by the programming error.

Keywords

Programming errors, security problems

Alkusanat

Haluan kiittää työni valvojaa ja ohjaajaa professori Juha Tuomista diplomityön tähän kuntoon auttamisesta. Lisäksi haluan kiittää häntä siitä, että hän on ollut mahdollistamassa diplomi-insinööriksi opiskelun Lahdesta käsin, mikä on tehnyt mahdolliseksi tämänkin lopputyön tekemisen. Haluan kiittää myös Jarkko Rapikistoa oikoluvusta sekä työkavereita ja ystäviä korjausvinkeistä.

Lahdessa 20.9.2004

Janne Parkkonen

SISÄLLYSLUETTELO

1 JOHDANTO 1

1.1 YLEISTÄ 1

1.2 TAVOITE 3

1.3 TUTKIMUSMENETELMIÄ 3

1.4 RAJAUS 3

1.5 DIPLOMITYÖN SISÄLTÖ 4

2 TIETOTURVA YLEISESTI 5

2.1 JOHDANTO 5

2.2 TIETOTURVAN MÄÄRITTELY 5

2.2.1 Tietoturvan elementit 5

2.2.2 Tietoturvan perusperiaatteita 6

2.3 TIETOTURVA SOVELLUSKEHITYKSESSÄ 7

2.3.1 Sovelluksen suunnitteluperiaatteet 7

2.3.2 Hyökkääjän etu ja puolustajan ongelma 8

2.3.3 Kaikki syöte on pahaa syötettä 8

2.4 TIETOTURVAVÄLIKOHTAUKSET JA NIIDEN AIHEUTTAMAT KUSTANNUKSET ... 9

2.4.1 Tietoturvavälikohtaukset 9

2.4.2 Tietoturvan kustannukset 10

2.5 YHTEENVETO 12

3 TIETOTURVAUHAN ANALYSOINTI JA MALLINTAMINEN 13

3.1 JOHDANTO 13

3.2 TURVASUUNNITTELUN HYÖDYT 13

3.3 MALLINNUSPROSESSI 14

3.4 JÄRJESTELMÄN KÄYTÖSSÄ OLEVIEN RESURSSIEN TUNNISTAMINEN 15

3.5 ARKKITEHTUURIN YLEISKUVAUKSEN LUOMINEN 15

3.6 SOVELLUKSEN OSITTAMINEN 17

3.7 UHKIEN TUNNISTAMINEN 19

3.8 UHKIEN DOKUMENTOINTI 22

3.9 UHKIEN JÄRJESTÄMINEN 23

3.10 YHTEENVETO 23

4 TIETOTURVAONGELMIA AIHEUTTAVAT OHJELMOINTIVIRHEET 25

4.1 JOHDANTO 25
4.2 PUSKURIYLIVUODOT 25
4.2.1 Kuvaus puskurin ylivuodoista 25
4.2.2 Pinon ylivuoto 26
4.2.2.1 Kuvaus pinon ylivuodosta 26
4.2.2.2 Esimerkki pinon ylivuodosta 29
4.2.2.3 Parannusehdotuksia pinon ylivuotoon 34
4.2.3 Keon ylivuoto 34
4.2.3.1 Kuvaus keon ylivuodosta 34
4.2.3.2 Esimerkki keon ylivuodosta 35
4.2.3.3 Parannuskeinoja keon ylivuotoon 37
4.3 YLEISIMMÄT WEB-HYÖKKÄYKSET 38
4.3.1 Johdanto 38
4.3.2 HTTP lyhyesti 38
4.3.3 Web-lomakkeiden ja keksien arvojen väärentäminen 41
4.3.3.1 Kuvaus 41
4.3.3.2 Esimerkki lomakkeen väärentämisestä 42
4.3.3.3 Esimerkki keksin väärentämisestä 46
4.3.3.4 Parannuskeinoja 50
4.3.4 Cross-site scripting ja selaimen ohjelmointi 50
4.3.4.1 Kuvaus 50
4.3.4.2 Esimerkki 51
4.3.4.3 Parannuskeinoja 53
4.3.5 SQL-injektiot 53
4.3.5.1 Kuvaus 53
4.3.5.2 Esimerkki 54
4.3.5.3 Parannuskeinoja 56
4.3.6 Palvelunestohyökkäys 56
4.3.6.1 Kuvaus 56
4.3.6.2 Esimerkki prosessorin kuormittamisessa 57

4.3.6.3 Parannuskeinoja 59
4.3.7 Verkkokuuntelu ja väärennökset 60
4.3.7.1 Kuvaus 60
4.3.7.2 Esimerkki http:n tunnistautumisesta 60
4.3.7.3 Parannuskeinoja 62
4.4 YHTEENVETO 62
5 YHTEENVETO 63
6 POHDINTA JA JOHTOPÄÄTÖKSET 64
7 SOVELLUSKEHITYKSEN TULEVAISUUS 65
LÄHTEET 67
LIITTEET 69

Lyhenteitä ja termejä

Bof	Puskurinylivuoto (engl. buffer overflow) tapahtuu, kun kopioidaan muistiin tietoa enemmän kuin mitä sille on varattu tilaa.
Cookie	Eväste, ts. keksi, jonka avulla voidaan säilyttää tilaa käyttäjän liikkuessa www-sivuilla.
DDoS	Hajautettu DoS (engl. Distributed DoS) (kts. DoS)
DoS	Denial Of Service, palvelunestohyökkäys, jolla pyritään kuormittamaan kohdejärjestelmä niin, ettei se ole muiden käytettävissä ([1], s. 510)
Debugger	Apuohjelma, jonka avulla sovelluskehittäjä voi jäljittää omassa sovelluksessaan olevia virheitä helpommin
Exploit	Tietoturvaongelmaa hyväksikäyttävä ohjelmakoodi ([2], s. 4)
Hakkeri	Henkilö (engl. hacker), joka pyrkii mahdollisimman tehokkaiden sovellusten tekemiseen tai joka tutkii sovellusten toimintaa pintaa syvemmältä ([3], s. 8)
HTTP	Hypertext Transfer Protocol määrittelee, kuinka tiedonsiirto tapahtuu World Wide Webissä.
HTML	Hypertext Markup Language määrittelee, kuinka www-sivu näytetään selaimessa.
Incident	Tietoturvavälikohtaus, jossa rikotaan tietoturvaperiaatetta tai sääntöä suorasti tai epäsuorasti [4]
IP-osoite	Internet Protocol-osoite on koneella oleva tunnisteosoite, jota käytetään koneiden välisessä liikennöinnissä toisen koneen tunnistamiseksi.

LIFO	Last In First Out –tietorakenne, jossa viimeiseksi rakenteeseen tullut data on ensimmäisenä lähtemässä pois, kun rakenteesta halutaan poistaa dataa ([2], s. 5)
Sequence diagram	Viestiyhteyskaavio kuvaa dynaamista yhteistyötä joidenkin olioiden välillä. Olioiden viestintää kuvataan tietyssä tilanteessa, ja se kuvataan kronologisessa järjestyksessä tapahtuvilla viesteillä. ([5], s. 18)
SSL	Secure Sockets Layer on mekanismi, jolla voidaan salata asiakkaan ja palvelimen välinen liikenne HTTP:ssä.
Stack-based bof	Pinoon kohdistuva puskurin ylivuoto. ([6], s. 131)
Static bof	Kts. Stack-based bof ([6], s. 131)
TLS	Transport Layer Security on SSL:stä kehitetty uudempi versio tietoliikenteen salaamiseen
URL	Uniform Resource Locator on yksilöivä osoite tiedostolle internetissä.
Use case diagram	Käyttötapauskaavio kuvaa järjestelmän tarjoamia palveluita ulkopuolisen toimijan kannalta. Toimija voi olla ihminen tai toinen järjestelmä, joka kommunikoi järjestelmän kanssa. ([5], s. 13)
XSS	Cross-Site Scripting on hyökkäys, jolla pystytään suorittamaan käyttäjän selaimessa ohjelmakoodia hänen ollessaan yhteydessä luotettuun web-sivustoon. ([7], s. 26-27)

1 Johdanto

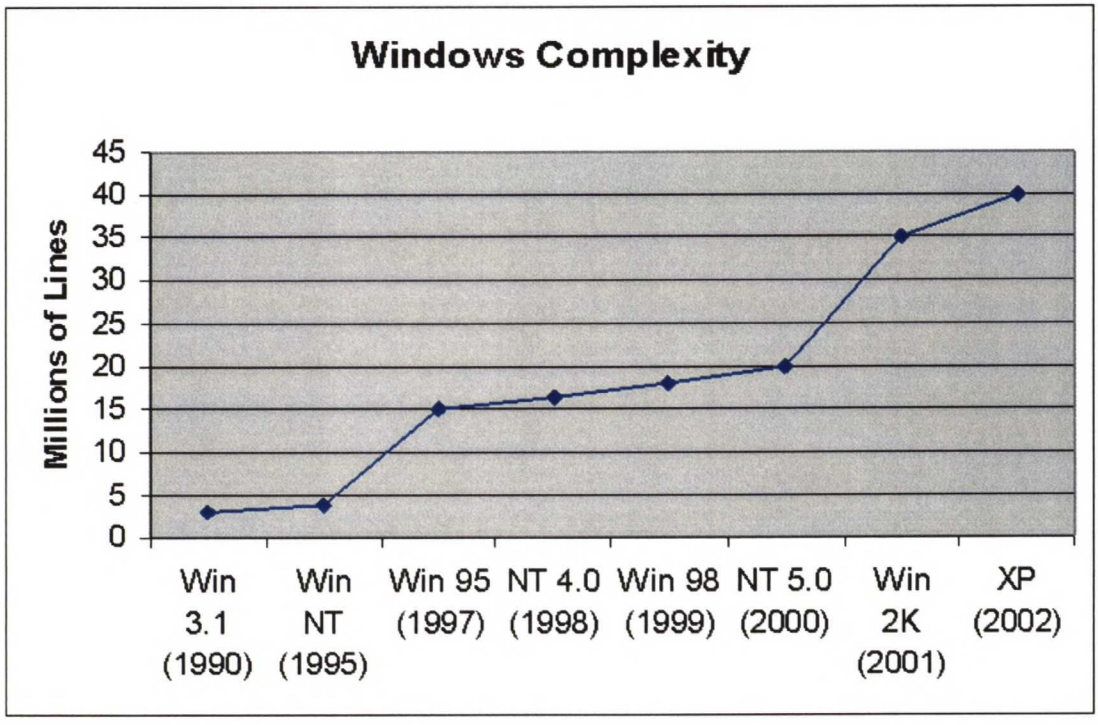
1.1 Yleistä

Nykypäivänä tietoturva-sanaa ei juuri kukaan pysty välttämään kuulemasta. Tietoturvasta puhutaan paljon uutisissa ja päivän lehdissä, milloin syyn ollessa leviävä virusepidemia ja milloin pelko omien pankkisalaisuuksien leviämisestä muiden ihmisten tietoon. Keskustelua käydään kuitenkin niin korkealla tasolla, että harva ihminen todella tietää mitä hän voisi tehdä toisin, jotta hänen kohdallaan tietoturva toteutuisi paremmin. Keskustelu saattaa aiheuttaa myös pelkoa uusien asioiden opettelussa ja käyttöönotossa, kun todellisuudessa suurin osa ihmisistä ei voi tehdä asioille yhtään mitään. Heidän on vain tyydyttävä olemassa oleviin käyttöjärjestelmiin ja ohjelmiin, vaikka he tietävät niissä olevan tietoturvaongelmia.

Tietokoneiden ja ohjelmien käyttö jokapäiväisessä elämässä on levinnyt lähes kaikkialle. Kaupat, pankit, vakuutusyhtiöt ym. nojaavat tietoteknisiin ratkaisuihin, eikä niillä ole varaa järjestelmien käyttökatkoksiin. Järjestelmien tulee olla käytettävissä koko ajan, ja niiden tulee toimia moitteettomasti myös suuren kuorman alla. Tämä asettaa kovat vaatimukset tietojärjestelmille ja niiden tärkeimmille osa-alueille, sovelluksille. Sovelluksissa ei saa olla virheitä, jotka aiheuttaisivat ongelmia järjestelmän käytettävyyden suhteen tai liiallisen kuormituksen, joka aiheuttaisi muille järjestelmän käyttäjille ongelmia. Esimerkkinä ruokatavarakauppa, jossa syntyy tapahtumarivejä noin 350 000 päivässä, jolloin tunnin käyttökatkoskin aiheuttaa toimintojen kasautumisen ([8], s. 14). Toisena esimerkkinä pankkien mikro- ja puhelinpalveluna tehtyjen tapahtumien määrä, joka oli vuonna 2003 160 miljoonaa [9]. Pankkien toiminnan kannalta tietoturva on ratkaiseva tekijä, jotta luottamus asiakkaan ja pankin välillä säilyisi.

Sovelluksien koot kasvavat jatkuvasti suurin harppauksin. Kun aikaisemmin sovellukset olivat rajoittuneita ja tarjosivat rajatun määrän käytettävissä olevia ominaisuuksia, nykyään sovellukset ovat mammuttimaisia ja pyrkivät tekemään kaiken, mitä käyttäjä voi vain ikinä haluta tehtävän. Esimerkkinä voi ol-

la vaikka tekstinkäsittely, jossa aikaisemmin käytettiin tekstinkirjoitukseen, oikolukuun, tulostukseen ym. aina eri ohjelmia. Nykyään koko ketjua hallitsee yksi ainoa ohjelma. Tämä asettaa ohjelmalle kovat vaateet siitä, että koko toimintaketju saadaan vietyä läpi kunniallisesti ja että käyttäjä saa halutut toiminnot suoritettua. Tämä luonnollisesti tarkoittaa sitä, että myös sovelluksen tekemiseen tarvittava lähdekoodimäärä kasvaa toimintojen lisääntyessä. Mikä koskee ohjelmien lisäksi myös käyttöjärjestelmiä. Kun 10-15 vuotta sitten käyttöjärjestelmässä oli kolme miljoonaa koodiriviä, nykyään niitä arvioidaan olevan jopa 40 miljoonaa ([10], s. 7). *Kuva 1* havainnollistaa koodirivien määrän kasvun Windows-käyttöjärjestelmissä.



Kuva 1: Windows-käyttöjärjestelmien koodirivien määrät versioittain ([10], s. 7)

Linuxin lähdekoodipaketin koko on vastaavasti kasvanut viidestä megatavusta (versio 1.0) noin pariin sataan megatavuun (versio 2.6.6) 10 vuoden aikana. Nämä kovat kasvulukemat ovat johtaneet siihen, että enää ei kukaan yksittäinen henkilö pysty hallitsemaan sovelluksen tai käyttöjärjestelmän kaikkia osa-alueita, vaan joudutaan turvautumaan monen ihmisen osaamiseen sovelluksen

kehityksessä. Osiin jakamisella saavutetaan se etu, että useat henkilöt voivat työskennellä samanaikaisesti ongelman kimpussa. Huonona puolena on, että joudutaan mahdollisesti turvautumaan rajapintamäärityksiin, jotta saadaan osat koottua myöhemmin yhteen. Rajapintaa suunniteltaessa ei aina kiinnitetä tarpeeksi huomiota siihen, ketkä kaikki itse asiassa voivat kutsua kyseistä rajapintaa. Varsinkin käyttöjärjestelmissä ja erilaisissa komponenteissa syntyy vakava tietoturvaongelma, kun rajapinnan yli tulee vääriä tai tarkoituksella vääriksi arvoiksi muodostettuja parametreja, jotka voivat saada sovellusohjelmoijan kannalta epätoivottuja toimintoja aikaan. Suuremman kokonaisuuden hallinta on haastavaa myös siksi, koska on kommunikoitava useiden henkilöiden kanssa, ennen kuin pystytään tekemään arvioita itse kokonaisuudesta.

Sovellusohjelmoijalle kannalta ongelmia tuottaa myös internet, joka on pullollaan erilaisia sovellusten hyväksikäyttöjä (engl. exploit), joilla voidaan esim. kaataa sovellus, saada pääkäyttäjän käyttöoikeudet, suorittaa komentoja etänä jne, mutta ohjeita, joilla voidaan estää hyväksikäytöt, on vähän. ([6], s. 128)

1.2 Tavoite

Tämän työn tavoitteena on käsitellä tietoturvaongelmia ja niihin johtavia syitä. Suurimmat syyt tietoturvaongelmien takana ovat huolimattomuus tai osaamattomuus. Kun tietoturvaongelmiin johtavat ongelmat ymmärretään, niihin pystytään myös keksimään korjaus- ja parannusehdotuksia, jotka johtavat parempaan lähdekoodin ja sitä kautta parempiin ja tietoturvallisempiin sovelluksiin.

1.3 Tutkimusmenetelmiä

Tutkimuksen apuna käytetään tietoturvaan ja erityisesti tietoturvaohjelmointiin liittyviä teoksia sekä tietoturvaongelmia aiheuttavia hyväksikäyttöohjelmien lähdekoodeja. Näiden käsittelyllä pyritään antamaan kokonaiskuva tämänhetkisistä ns. kuumista tietoturvaongelmien aiheuttajista.

1.4 Rajaus

Työssä ei käsitellä tietoturvaan yleisesti vaikuttavia toimenpiteitä, kuten käyttäjän tai ylläpitäjien toimia, eikä muitakaan seikkoja, jotka vaikuttavat tietoturvaan kokonaisuutena, kuten laitteisto ja tietoliikenne.

Työssä ei myöskään neuvota, miten voidaan tehdä mahdollisimman hyvä tietoturva-aukkoja hyödyntävä sovellus, vaan pyritään käsittelemään asia siten, että voidaan löytää omissa sovelluksissa olevat tietoturva-aukot.

Tietoturvaongelmien käsittely on rajattu kahteen käyttöjärjestelmään Windowsiin ja Linuxiin. Muista käyttöjärjestelmistä ei anneta esimerkkejä, vaikkakin moni niistä toimisi joko sellaisenaan tai hieman muutettuna myös erilaisissa kohdejärjestelmissä.

1.5 Diplomityön sisältö

Sisältö alkaa kappaleella 2, jossa käydään yleisellä tasolla läpi tietoturvaan liittyviä asioita. Siinä käsitellään tietoturvaongelmien seurauksia, kuten kustannuksia ym. Kappaleessa 3 käydään läpi tietoturvauhkan analysointi ja mallintaminen, joiden avulla voidaan suunnitella mahdollisimman turvallisia sovelluksia. Kappaleessa 4 käydään läpi tarkasti yleisimpiin tietoturvaongelmiin johtavat ohjelmointivirheet. Ongelmia käsitellään esimerkkien avulla, jotta ongelmat havainnollistuisivat helpommin. Esimerkkejä käsitellään sekä Windows- että Linux-ympäristössä. Kappaleessa 5 tehdään yhteenveto ohjelmointivirheistä sekä katsotaan tulevaisuuteen ennakoiden mahdollisia parannuskeinoja, joilla kappaleessa 4 esitetyistä ongelmista päästäisiin eroon.

2 Tietoturva yleisesti

2.1 Johdanto

Tässä luvussa käydään läpi tietoturvaperiaatteita ja yleisiä ongelmia, jotka liittyvät tietoturvaan sekä sovellusten tehokkaaseen ja turvalliseen kehittämiseen. Luvun alussa määritellään lyhyesti tietoturvaan kuuluvat elementit ja käydään läpi perusperiaatteita, jotka kuvaavat tietoturvaan liittyviä haasteita. Tämän jälkeen käsitellään tietoturvaa sovelluskehityksen näkökulmasta nostaen esiin seikkoja, jotka kannattaa huomioida sovelluksen suunnitteluvaiheessa tai jotka tulee tiedostaa sovellusta suunniteltaessa. Luvun lopuksi käsitellään tietoturvavälikohtausten määrän kasvua ja niiden aiheuttamia kustannuksia.

2.2 Tietoturvan määrittely

2.2.1 Tietoturvan elementit

Tietoturvan voidaan katsoa koostuvan kuudesta elementistä:

1. Luottamuksellisuus (engl. confidentiality)
2. Eheys (engl. integrity)
3. Saatavuus (engl. availability)
4. Todennus (engl. authentication)
5. Pääsynvalvonta (engl. access control, authorization)
6. Kiistämättömyys (engl. non-repudiation)

Luottamuksellisuus tarkoittaa sitä, että tiedot ovat vain niiden henkilöiden käytettävissä, joilla on oikeus käsitellä tietoja. Luottamuksellisuus saavutetaan usein käyttämällä salausta (engl. encryption), jolla saadaan tiedot suojattua ulkopuolisilta. *Eheys* taas varmistaa sen, ettei ulkopuolinen taho pääse muuttamaan tietoja. Tämä pystytään varmistamaan esim. tarkistussummien (engl. checksum) avulla. *Saatavuus* takaa palveluiden käytettävyyden henkilöille, jotka ovat niihin oikeutettuja. *Todennuksessa* varmistutaan tietoturvaolion - joka voi olla henkilö, laite tai ohjelmisto - aitoudesta, eli siitä, että olio on todella se mikä väittää olevansa. Todennus mahdollistaa luottamuksellisuuden. *Pääsynvalvonta* vastaa siitä, että vain henkilöt, joilla on oikeus käyttää järjestelmää tai järjestelmän kyseistä resurssia, saavat niin tehdä. Pääsynvalvontaan liittyy myös käytön seuranta (engl. audit), jolla pidetään kirjaa tietoturvaolion

suorittamista toimista järjestelmässä. *Kiistämättömyys* varmistaa sen, ettei tietoturvaolio voi kiistää tehneensä operaatiota, joka on todellisuudessa tehty. Esimerkkinä henkilö, joka kiistäisi myöhemmin tilanneensa tuotteita, vaikka näin olisikin tehnyt. Kiistämättömyys on noussut tärkeään asemaan sähköisen kaupankäynnin lisääntyessä.
([11], s. 22-27; [12], s. 4-8; [13], s. 7-12, 23-25)

2.2.2 Tietoturvan perusperiaatteita

Tietoturvaperiaatteita voidaan tiivistää sanonnoiksi, jotka kuvaavat tietoturvaan liittyviä ongelmia. Ne kuvaavat myös ongelmia ja haasteita, joihin sovelluskehittäjät törmäävät sovellusten suunnittelu, ja -toteutusvaiheissa. Sanonnat ovat seuraavat:

1. Mukavuus * tietoturva = vakio
2. Usko hyvästä tietoturvasta on vaarallisempaa kuin tieto huonosta
3. Odota odottamatonta
4. Turvallisuutta ei voi lisätä jälkikäteen

Mukavuus * tietoturva = vakio

Sanonta kuvaa sitä ongelmaa, joka liittyy tietoturvaa parantavien ominaisuuksien ja sovellusten käyttömukavuuden väliseen yhteyteen. Tietoturvan edellytyksenä on usein monimutkaiset ja tiheään vaihtuvat salasanat sekä erilaiset turvallisuusasetusten määrittelyt, jotka teettävät sovellusta käyttävälle henkilölle lisää töitä. Tietoturvan asettamat, esim. laskennalliset vaatimukset, jotka vievät paljon aikaa, voivat myös turhauttaa käyttäjää. Langaton lähiverkko on hyvä esimerkki tästä sanonnasta, koska sen valtti on vapaa liikkuminen, mikä mahdollistaa myös liikenteen salakuuntelemisen.
([11], s. 43-44; [12], s. 9-10)

Usko hyvästä tietoturvasta on vaarallisempaa kuin tieto huonosta

On tärkeää tietää todellinen tietoturvan taso, jottei tuudittauduta hyväuskoisesti luulemaan asioiden olevan paremmin kuin mitä ne todellisuudessa ovat. Tietämys tietoturvatasosta vaikuttaa myös käyttäjiin, jotka toimivat

varovaisemmin huonossa tilanteessa ja huolimattomasti hyvässä tilanteessa. ([11], s. 46)

Odota odottamatonta

On helppo varautua ongelmiin, jotka tiedetään etukäteen. Ongelmiin, joita emme edes vielä tiedä olevan, on äärimmäisen hankala varautua. Tämä luo sovelluskehittäjille melkoisen käytännön ongelman, koska uhat muodostuvat yleensä juuri odottamattomista asioista, kuten kappaleessa 2.3.2 myös todettiin. Lisäksi sovellusohjelmoijan kannattaa varautua siihen, että hyökkääjä saa käsiinsä samat tiedot kuin mitä tekijöillä on. Ei siis saa luottaa siihen, että vastapuolella ei olisi samoja tietoja käytettävissä, kuin mitä itsellä on (engl. security through obscurity).
([11], s. 47, 66)

Turvallisuutta ei voi lisätä jälkikäteen

Kun sovellukseen lisätään jälkikäteen turvallisuutta tuovia ominaisuuksia, saavutetaan vain suojakerros olemassa oleville ominaisuuksille. Sillä ei pystytä korvaamaan sovelluksen suunnitteluvaiheessa syntyneitä, ominaisuuksiin vaikuttavia turvallisuusratkaisujen puutteita. Turvallisuuden, kuten muiden ominaisuuksienkin, lisääminen jälkikäteen on työlästä ja kallista. Turvallisuuden lisääminen saattaa myös rikkoa olemassa olevan toteutuksen, mikä voi teettää paljon ylimääräistä työtä.
([6], s. 39)

2.3 Tietoturva sovelluskehityksessä

2.3.1 Sovelluksen suunnitteluperiaatteet

Sovelluksen suunnittelussa kannattaa käyttää kolmea periaatetta:

1. Turvallisuus suunnittelussa (engl. secure by design)
2. Turvallisuus oletuksena (engl. secure by default)
3. Turvallisuus sovelluksen hallinnoinnissa (engl. secure by deployment)

Turvallisuus suunnittelussa tarkoittaa sitä, että suunnitellaan sovellus kokonaisuutena mahdollisimman turvalliseksi. Pyritään käyttämään apuna menetelmiä, joilla pystytään parantamaan sovelluksen tietoturvaa, kuten tietoturvauhan ana-

lysointia ja mallintamista (lisää luvussa 3). *Turvallisuus oletuksena* tarkoittaa sitä, että tuotteen tietoturva on mahdollisimman hyvä heti asennuksen jälkeen. Tähän päästään minimoimalla käynnistettävien palveluiden lukumäärä ja vähentämällä käytössä olevien ominaisuuksien määrää. Oletuksena käytetään mahdollisimman vähiä käyttöoikeuksia, jotta hyökkäyksen onnistuessa hyökkääjä saisi mahdollisimman heikot oikeudet käyttöönsä. *Turvallisuus sovelluksen hallinnoinnissa* tarkoittaa sovellusta, joka on mahdollisimman helppo asentaa ja ylläpitää. Tällöin ylläpitäjät pystyvät poistamaan käytössä olevia haavoittuvia ominaisuuksia ja pitämään sovelluksen mahdollisimman turvallisena. Lisäksi turvallisen hallinnoinnin saavuttamiseksi tulee sovelluksen ohjeiden olla niin monipuoliset, että käyttäjät pystyvät käyttämään sovellusta turvallisesti.

([6], s. 51-54)

2.3.2 Hyökkääjän etu ja puolustajan ongelma

Kun sovellus on tehty ja toimitettu, sovelluksen kehittäjät pystyvät hyvin vähän vaikuttamaan siihen, mitä sillä tai mitä sille tehdään. Sovellus on näin ollen jatkuvasti puolustustilassa mahdollisia hyökkäyksiä vastaan. Hyökkääjä vastaavasti voi hankkia sovelluksen itselleen ja etsiä siitä tietoturva-aukkoja kenenkään tietämättä. Tätä kuvaakin hyvin tietoturvasanonta: ”hyökkääjän etu ja puolustajan ongelma”. Väitettä voidaan perustella seuraavin kohdin:

1. Puolustajan pitää puolustaa kaikkia pisteitä, mutta hyökkääjä voi valita heikoimman pisteen
2. Puolustaja voi puolustaa vain tunnettuja hyökkäyksiä vastaan, kun taas hyökkääjä voi tutkia uusia hyökkäyksiä
3. Puolustajan pitää olla jatkuvasti puolustamassa, mutta hyökkääjä voi valita hyökkäyshetkensä
4. Puolustajan pitää pelata sääntöjen mukaan, mutta hyökkääjä voi käyttää likaisia keinoja

([6], s. 19-21)

2.3.3 Kaikki syöte on pahaa syötettä

Suurin osa sovelluksissa olevista tietoturva-aukoista johtuu huonosta syötteen tarkistuksesta, kuten myös luvussa 4 nähdään. Sovellukset luottavat syöttee-

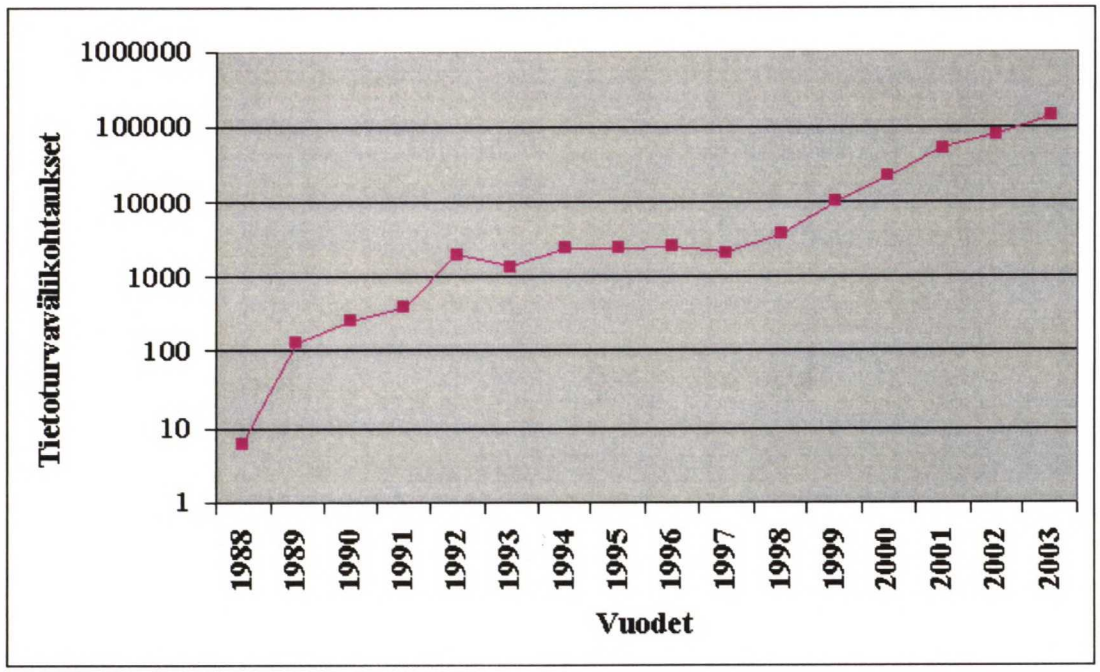
seen tai sitä ei tarkisteta riittävän tehokkaasti, jolloin vihamielinen syöte saa aikaan tietoturvaongelman. Siksi kaikkea sovellukselle syötettyä dataa tulisi pitää vihamielisenä ennen kuin on varmistettu sen olevan sallittua. Data tulisi myös tarkastaa aina, kun sitä siirretään luotettavan ja epäluotettavan ympäristön välillä. Luotettavan datan on sovelluskehittäjä itse varmistanut, tai sen on tehnyt joku toinen, johon sovelluskehittäjä luottaa. Epäluotettavaksi luetaan kaikki muu data. Näin vältetään tilanne, jossa luullaan syötteen tarkistuksen suoritettavan jossain muualla, vaikka se pitäisi hoitaa itse. Syöte pitää myös tarkistaa, vaikka sovellusta tiedetään käyttävän luotettava henkilö, koska hänkin saattaa tehdä syöttövirheen, jonka seurauksena sovellus voi pahimmillaan kaatua.

([6], s. 341)

2.4 Tietoturvavälikohtaukset ja niiden aiheuttamat kustannukset

2.4.1 Tietoturvavälikohtaukset

Tietoturvavälikohtausten (engl. incidents) määrä on kasvanut huimasti vuosien varrella, kuten *kuvasta 2* näkyy.



Kuva 2: Tietoturvavälikohtausten määrä vuodessa logaritmisella asteikolla [14]

Kuvassa 2 oleva y-akseli on logaritminen kuvastaen välikohtausten määrän selvää kasvua viimeisen 10 vuoden aikana.

Vuosi	1988	1989	1990	1991	1992	1993	1994	1995
TTVK	6	132	252	406	1992	1334	2340	2412
Vuosi	1996	1997	1998	1999	2000	2001	2002	2003
TTVK	2573	2134	3734	9859	21756	52658	82094	137529

Taulukko 1: Tietoturvavälikohtausten määrä vuodessa [14]

Taulukossa 1 näkyy miten välikohtausten määrä on reilusti yli kymmenkertaistunut vuodesta 1999 vuoteen 2003!

2.4.2 Tietoturvan kustannukset

Tietoturvan aiheuttamat kustannukset leviävät sovelluksen kaikkiin vaiheisiin, mutta ne ovat sitä pienemmät mitä aikaisemmin virheet saadaan korjattua. Vahingon torjunta on siis halvempaa kuin vahingon korjaaminen ([11], s. 45). Microsoftin tietoturvayksikkö arvioi yhden tietoturvakorjauksen - josta joudu-

taan julkaisemaan tietoturvailmoitus (engl. security bulletin) - maksavan peräti 100 000 $ ([6], s. 11). Korjauksen kulut kasvavat suureksi, koska korjaukseen vaaditaan useita työvaiheita, joista kukin vie aikaa ja rahaa:

- Tietoturvaongelman korjauksen koordinointi
- Tietoturvavirheen etsiminen lähdekoodista
- Virheen korjaaminen
- Virheenkorjauksen toimivuuden testaaminen
- Virhekorjauspaketin toimivuuden testaaminen
- Kansainvälisten versioiden luominen ja testaaminen
- Korjauksen digitaalinen allekirjoitus, jos sellainen on käytössä
- Korjauksen siirto verkkoon yleiseen jakeluun
- Korjaukseen liittyvän dokumentaation tuottaminen
- Yrityksen saaman huonon julkisuuden hoitaminen
- Korjauksen levityksestä johtuva lisäkaistan kulutus verkkosivustoilla
- Tuottavuuden menetys, kun henkilöt ovat sidottuina vanhojen virheiden korjaukseen, vaikka pitäisi tehdä jo uusia ominaisuuksia
- Asiakkaiden kustannukset heidän testatessaan ja asentaessaan korjauksia omiin ympäristöihinsä. Mahdollisesti myös lisäkustannuksia, kun joudutaan tekemään sovelluksia, jotka etsivät verkoista tietokoneita, joihin ei ole vielä asennettu päivityksiä.
- Potentiaalinen tulojen menetys asiakkaiden lakatessa käyttämästä valmistajan sovelluksia tai lykätessä niiden hankintoja tietoturvaongelmien takia

([6], s. 10)

Sovelluksia käyttävien yritysten ja henkilöiden tietoturvakustannuksiin kuuluvat myös erilaiset tietoturvatuotteet, kuten virustorjunta ja palomuurit. Tuotteita valmistavat tietoturvayhtiöt, jotka pyrkivät estämään tietoturva-aukkoja hyödyntävien sovellusten, kuten matojen (engl. worm), leviäminen. Tietoturvayhtiöille on siis hyötyä virus- tai matoepidemioista, jotka aiheutuvat sovelluksissa olevista tietoturva-aukoista, koska pelko uusista epidemioista pakottaa ihmiset hankkimaan tietoturvaa ulkoisesta tuotteesta. Silloin tällöin suojaus kuitenkin pettää, jonka seurauksena virus tai mato voi tuhota erittäinkin tärkei-

tä tietoja. Tärkeiden tietojen palauttaminen vahingon jäljiltä voi osoittautua mahdottomaksi tai hyvin kalliiksi.
([11], s. 28-29, 257)

2.5 Yhteenveto

Luvussa käsiteltiin tietoturvaa yleisesti tietoturvan määrittelystä sovelluskehityksen näkökulmaan. Määrittelyosuudessa käytiin läpi elementit, joista tietoturva koostuu. Seuraavassa luvussa käsitellään tietoturvauhan analysointia ja mallinnusta, jonka tarkoituksena on toteuttaa mahdollisimman tehokkaasti tässä luvussa esitetyt tietoturvan osatekijät.

3 Tietoturvauhan analysointi ja mallintaminen

3.1 Johdanto

Luvussa käsitellään menetelmää, jonka avulla pystytään mallintamaan tietoturvauhkia formaalilla tavalla. Menetelmän tavoitteena on saada sovellusohjelmoijat huomaamaan sovellustensa haavoittuvimmat osa-alueet ja saada heidät valitsemaan sopivimmat menetelmät tietoturvauhkien pienentämiseksi. Luku alkaa suunnittelun hyötyjen läpikäynnillä, jossa perustellaan miksi kyseistä menetelmää kannattaa ylipäätään käyttää. Tämän jälkeen esitellään mallinnusprosessi ja siinä olevat kuusi vaihetta. Seuraavaksi jokainen vaiheista käydään läpi selittäen siihen kuuluvat työvaiheet.
([6], s. 69; [15])

3.2 Turvasuunnittelun hyödyt

Turvallisia sovelluksia ei voida tehdä ennen kuin tiedetään sovellukseen kohdistuvat uhkatekijät. Suunnittelun ja mallintamisen tavoitteena onkin tuoda esiin uhat, arvioida ne sekä vähentää niiden aiheuttamaa riskiä. Hyvin tehty malli tuo myös muita etuja, kuten:

- Malli auttaa ohjelmoijia ymmärtämään sovellustaan paremmin, koska he mallia rakentaessaan joutuvat miettimään tarkasti, kuinka sovellus toimii kyseisissä tilanteissa
- Malli helpottaa virheiden löytämistä, koska kriittinen sovelluksen arviointi nostaa esiin sovelluksessa olevia epäkohtia
- Mallin avulla löydetään monimutkaiset suunnitteluvirheet, joita ei todennäköisesti löydetä muilla keinoilla, kuten lähdekoodin lukemisella tai sovelluksen testaamisella
- Mallin avulla uudet ohjelmoijat pääsevät helpommin sovellukseen toimintaan kiinni, jolloin uusi henkilö pystyy tuottaviin työtehtäviin nopeammin kuin aikaisemmin
- Mallin avulla sovellukseen liittyvien toisten sovellusten tekijöiden on helpompi arvioida sovellusten yhteisiä uhkia sekä sovellusten välisessä rajapinnassa olevia uhkia

- Mallin avulla testaajat voivat myös testata sovellusta paremmin

([6], s. 70-71)

Mallin analysointi voi olla työlästä, mutta siitä saadut hyödyt ovat suuret. Mallin rakennusvaiheessa löydetty sovelluksen suunnitteluun liittyvän virheen korjaaminen on tässä vaiheessa paljon halvempaa kuin varsinaisessa implementaatiovaiheessa. Malli tulee myös pitää ajan tasalla, kun sovellukseen vaikuttavat olosuhteet muuttuvat.

([6], s. 71)

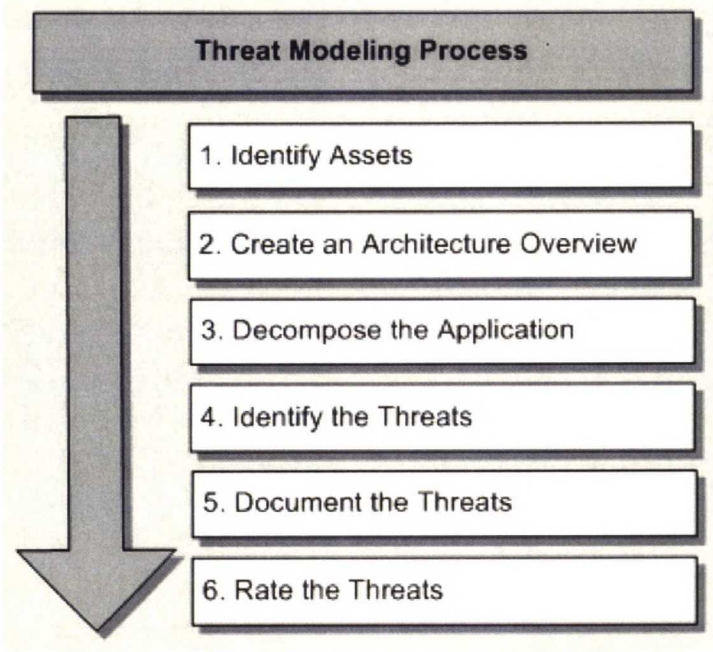
3.3 Mallinnusprosessi

Mallinnusprosessiin kuuluu seuraavat vaiheet:

1. Järjestelmän käytössä olevien resurssien tunnistaminen
2. Arkkitehtuurin yleiskuvauksen luominen
3. Sovelluksen osittaminen
4. Uhkien tunnistaminen
5. Uhkien dokumentointi
6. Uhkien järjestäminen niiden vakavuuden perusteella laskevaan järjestykseen

Prosessia tulee iteroida muutaman kerran, koska kukaan ei pysty ensimmäisellä kerralla määrittelemään kaikkia mahdollisia uhkia. Prosessin vaiheet näkyvät *kuvassa 3*.

([6], s. 71-72; [15])



Kuva 3: Tietoturvauhkien mallinnuksen prosessikaavio [15]

Kappaleissa 3.4 – 3.9 käydään tarkemmin läpi jokainen prosessin vaihe.

3.4 Järjestelmän käytössä olevien resurssien tunnistaminen

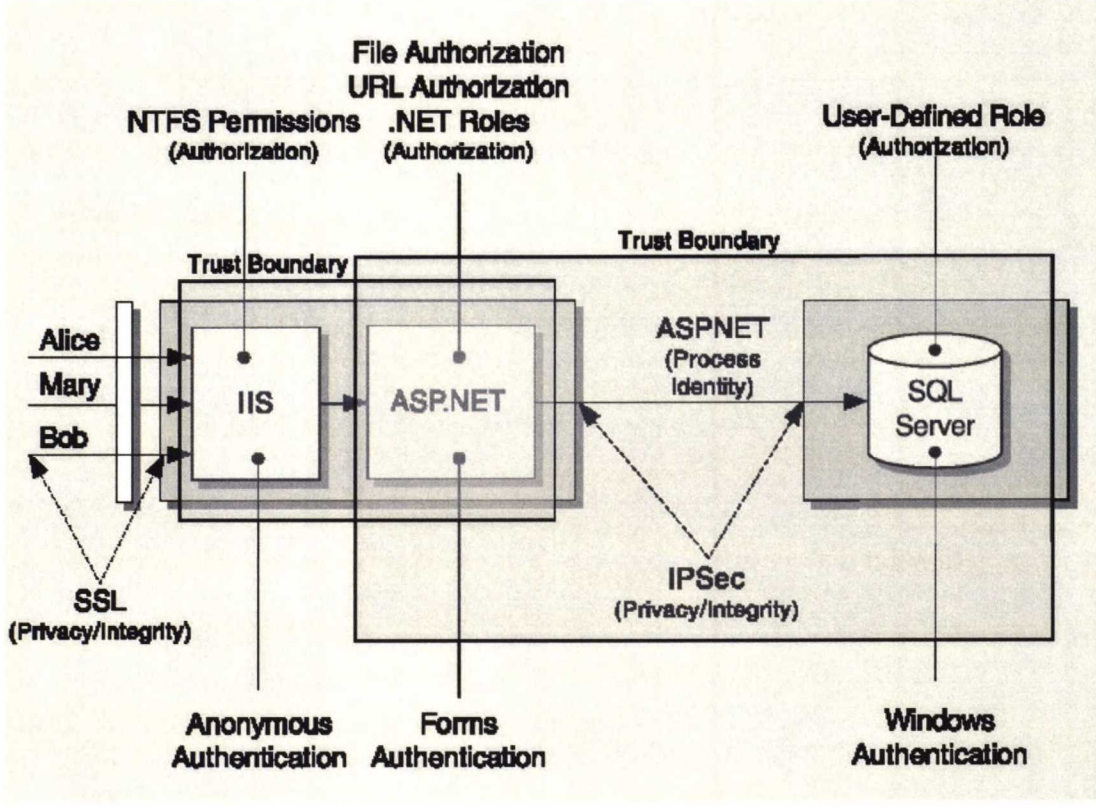
Järjestelmän resurssi voi olla tietokannan data, arkaluontoinen tieto tai yleisemmin tässä yhteydessä uhan alla oleva kohde. Jokainen resurssi, joka voi kiinnostaa hyökkääjää, tulee tunnistaa ja luetteloida. Näiden resurssien turvaamista arvioidaan myöhemmin uhkia käsiteltäessä.

([6], s. 87; [15])

3.5 Arkkitehtuurin yleiskuvauksen luominen

Arkkitehtuurin yleiskuvauksessa on tarkoitus kuvata sovelluksen toiminta, arkkitehtuuri, konfiguraatio sekä teknologiat, joita sovellus käyttää. Näitä arvioimalla voidaan löytää potentiaalisia haavoittuvaisuuksia suunnittelussa tai implementaatiossa. Yleiskuvaukseen kuuluu sovelluksen toiminnan läpikäynti, arkkitehtuuridiagrammi sekä teknologioiden tunnistaminen. Sovelluksen toiminnan läpikäynnissä tunnistetaan, mitä sovellus tekee ja miten se käyttää resurssejaan. Resurssien käsittelyssä huomioidaan erityisesti, miten sovellus pääsee käsiksi kyseisiin resursseihin. Sovelluksen toimintaa kuvataan käyttötapauskaavioilla (engl. use case diagram), jotka auttavat ymmärtämään miten

sovelluksen tulisi toimia. Tämä auttaa myös havaitsemaan mahdollisia keinoja sovelluksen väärinkäyttämiseen. Arkkitehtuuridiagrammi on korkean tason kuvaus sovelluksen osituksesta ja ryhmittelystä. Lisäksi se kuvaa alijärjestelmät ja fyysisen konfiguraation. Monimutkaisessa järjestelmässä arkkitehtuuridiagrammia kannattaa täydentää eri osa-alueista tehdyillä lisädiagrammeilla. Esimerkkinä arkkitehtuuridiagrammista on *kuva 4*.
[15]

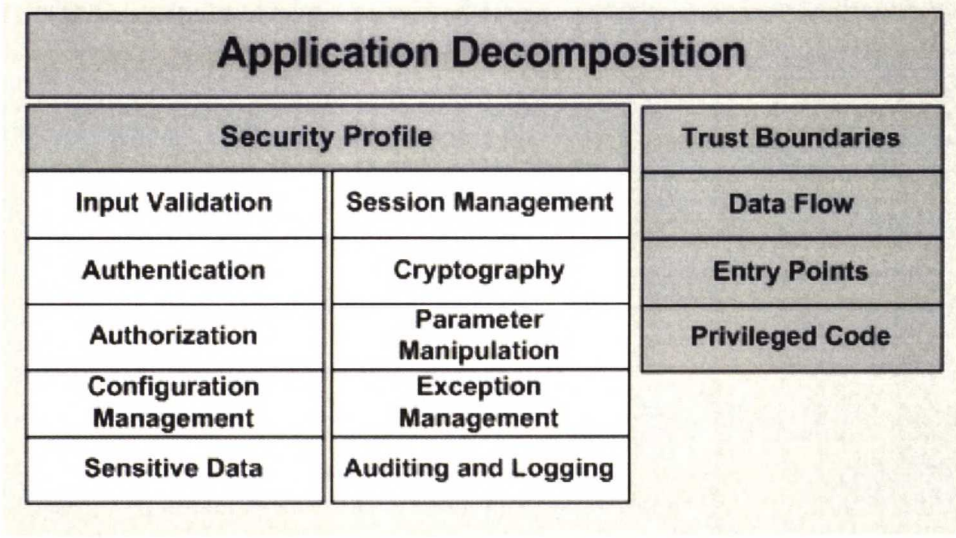


Kuva 4: Esimerkki arkkitehtidiagrammista [15]

Teknologioiden tunnistamisessa pyritään luetteloimaan ja kuvaamaan yksityiskohtaisesti kaikki teknologiat, joita sovelluksessa käytetään. Tämä helpottaa myöhemmin teknologiariippuvaisten uhkien arvioinnissa sekä uhkaa pienentävien menetelmien valitsemisessa. Teknologioita ovat esimerkiksi käytettävät tietokanta-alustat, siirtoyhteyskerroksen protokollat, käyttöjärjestelmät jne.
[15]

3.6 Sovelluksen osittaminen

Sovelluksen osittamisessa pyritään jakamaan sovellus turvallisuusprofiiliin, joka koostuu perinteisistä hyökkäysalueista. Tämän lisäksi määritellään sovelluksen luottamusrajat (engl. trust boundaries), tietovirtaukset sovelluksessa (engl. data flow), sovelluksen käyttöpisteet (engl. entry points) ja etuoikeutetun oikeuden käyttö (engl. privileged code). *Kuvassa 5* näkyy kohteet sovelluksen osittamisprosessille.
[15]



Kuva 5: Sovelluksen osittamisessa läpikäytävät osa-alueet [15]

Luottamusrajat tulee määritellä jokaisen sovelluksessa käytettävän resurssin ympärille (katso 3.4). Jokaiselle alijärjestelmälle tulee määritellä, luottaako alijärjestelmä ylemmän järjestelmän tai käyttäjän syötteeseen vai ei. Jos alijärjestelmä ei luota syötteeseen, tulee arvioida sitä, miten syötteen oikeudet voidaan todentaa (engl. authentication) sekä tarkistaa (engl. authorization). Täytyy myös arvioida luotetaanko kutsuvaan koodiin vai ei, jolloin myös kutsuva koodi tulee voida todentaa ja tarkistaa. On myös varmistettava, että kaikki sovelluksen käyttöpisteet ovat varmistettuja tietylle luottamusalueelle. Tällä luottamusalueella tulee tarkastaa täydellisesti luottamusalueen yli tuleva syöte. Luottamusrajojen analysoinnin voi aloittaa koodin näkökulmasta, jolloin arvioidaan koodi siltä osin kuin sovellus tarvitsee toimiakseen kyseisellä luotta-

musrajalla. Näiltä osa-alueilta arvioidaan rajoilla toimivien tahojen luottamuksellisuus. Palvelinten välisiä luottamusrajoja tulee myös arvioida eritoten käyttäjän todentamiseen ja käyttöoikeuksien tarkistamiseen liittyvissä tai datan siirtoon liittyvissä tapauksissa. Esimerkkinä palvelinten välisestä luottamusrajasta voisi olla sovelluspalvelimen ja tietokantapalvelimen välinen yhteys. [15]

Tietovirtojen analysointi kannattaa aloittaa korkealta tasolta, josta iteroimalla ositetaan järjestelmä alijärjestelmiin ja alijärjestelmät edelleen uusiin alijärjestelmiin. Erityisen tärkeitä ovat luottamusrajojen väliset tietovirrat, koska vastaanottavan järjestelmän tai komponentin tulee pitää vastaanottamiaan tietoja vihamielisenä ja suorittaa niille täydelliset tarkistukset ennen kuin niitä voidaan käyttää toiminnallisuuksien toteuttamiseen. Tietovirtojen kuvaamiseen voidaan käyttää DFD –kaavioita (engl. data flow diagrams). Niiden avulla voidaan kuvata formaalisti järjestelmän eri osa-alueita, niiden suhteita sekä tietovirtoja niiden välillä. DFD tehdään iteroimalla aloittaen sovelluksen tai järjestelmän kuvausdiagrammista, josta sitten poraudutaan alijärjestelmiin ja edelleen tarvittaessa alijärjestelmien alijärjestelmiin. DFD:n lisäksi voidaan käyttää viestiyhteyskaavioita (engl. sequence diagrams), joilla voidaan kuvata olioiden välistä kommunikointia ajan kuluessa ([5], s. 18).
([6], s. 73-81; [15])

Sovelluksen käyttöpisteet ovat myös hyökkäyksen käyttöpisteitä. Käyttöpiste voi olla esimerkiksi web-palvelin, joka kuuntelee asiakkaan pyyntöjä. Tämän lisäksi käyttöpisteitä voivat olla myös sisäiset komponentit. Täten tulee myös arvioida jokaisen komponentin kohdalta mahdollisuus, että hyökkääjä pystyy ohittamaan ensisijaisen käyttöpisteen ja syöttämään tietoja suoraan komponenttiin. Seuraavaksi jokaiselle sovelluksessa olevalle käyttöpisteelle tulee määritellä käytettävä todennuksen hallinta ja tiedon tarkistuksen taso. [15]

Etuoikeutetun oikeuden käyttöä tarvitaan, kun joudutaan käsittelemään tietoja tai resursseja, joita ei ole vähemmillä oikeuksilla saatavilla. Tällaisia resursse-

ja voivat olla tietokannat, levyt, hakemistot, tulostimet ja muut palvelut. Resurssit tulee arvioida tarkasti, jottei turhaan käytetä tarpeettoman laajoja oikeuksia. Etuoikeutetuilla oikeuksilla ajettavaan koodiin on suhtauduttava kriittisesti, jottei sen oikeuksilla päästä suorittamaan vihamielistä koodia.
[15]

Sovelluksen osittamisen lopuksi dokumentoidaan turvallisuusprofiili. Sen tarkoituksena on käydä läpi jokainen profiilin kategoria läpi ja arvioida siihen liittyvät uhat. Apuna kunkin kategorian uhkien arvioinnissa voidaan käyttää kysymyksiä, joiden avulla on helpompi arvioida uhkien vaikutuksia sovellukselle. Esimerkiksi syötteen tarkistuksessa (engl. input validation) voidaan kysyä tarkistetaanko kaikki sovelluksen saama syöte. Voidaan myös kysyä tarkistetaanko data aina kun sitä siirtyy luottamusrajojen välillä.
[15]

3.7 Uhkien tunnistaminen

Uhkien tunnistamisessa voidaan käyttää apuna kategorisointia, kuten STRIDE:ä. STRIDE koostuu seuraavista kategorioista:

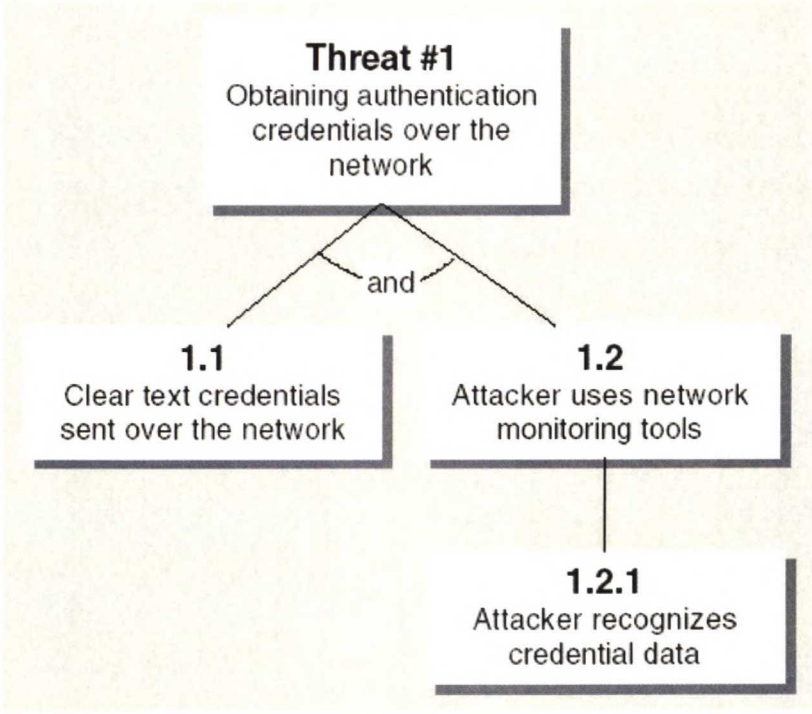
- Identiteetin väärentäminen (engl. **S**poofing identity)
- Datan väärentäminen (engl. **T**ampering with data)
- Operaation kieltäminen (engl. **R**epudiation)
- Tiedon paljastuminen (engl. **I**nformation disclosure)
- Palvelunesto (engl. **D**enial of service)
- Käyttöoikeuksien korotus (engl. **E**levation of privilege)

Toinen keino uhkien tunnistamiseen on käyttää apuna listaa (esim. [7], s. 18-42), johon on kerätty yleisimmät, esim. verkkoon, palvelimeen ja sovellukseen liittyvät uhat.
([6], s. 83-86; [7], s. 16-18; [15])

Identiteetin väärentämisessä hyökkääjä yrittää saada itselleen käyttöoikeudet järjestelmään käyttäen väärennettyä identiteettiä, joka voi esittää toista henkilöä tai palvelinta. Identiteetti voidaan väärentää varastamalla käyttäjän tunnus ja salasana, käyttämällä käyttäjän koneen tai palvelimen IP-osoitetta tai kaap-

paamalla verkkoliikenteessä kulkevia käyttäjätietoja. Kun hyökkääjä on onnistunut saamaan itselleen järjestelmän hyväksymän identiteetin, hän voi yrittää käyttöoikeuksien korotusta tai muita hyökkäyksiä. *Datan väärentämisessä* hyökkääjä muuttaa luvatta dataa, mikä voi tapahtua esimerkiksi muokkaamalla kahden koneen välistä liikennettä. *Operaation kieltämisessä* hyökkääjä kieltää tehneensä jotain, joka kuitenkin on tehty. Jos järjestelmä ei seuraa tarkasti siihen tehtyjä toimintoja, ei operaation kieltämistä voida helposti todistaa. *Tiedon paljastumisessa* hyökkääjä saa käsiinsä tietoja, joihin hänellä ei ole oikeuksia. Tiedot voivat olla henkilöiden henkilökohtaisia tietoja, tärkeitä tiedostoja, kahden tietokoneen välillä liikkuvaa dataa tai sovelluksessa käytettävän tietokannan yhteyden määrittävä merkkijono (engl. connection string). *Palvelunestossa* hyökkääjä pyrkii lamaannuttamaan järjestelmän tai sovelluksen niin, että se ei ole kenenkään muun käytettävissä. *Käyttäjäoikeuksien korotuksessa* hyökkääjä pystyy korottamaan identiteettinsä käyttöoikeuksia korkeammiksi, kuten esimerkiksi pääkäyttäjän tai järjestelmän oikeuksiksi.
([6], s. 84-85; [7], s. 16-17)

Jos uhkien tunnistamisessa käytetään jo tunnettujen uhkien listaa, muita sovelluksessa olevia potentiaalisia uhkia voi jäädä huomaamatta. Tämän takia tunnistamisessa kannattaa käyttää apuna myös hyökkäyspuita (engl. attack trees) ja hyökkäyskaavoja (engl. attack patterns). Hyökkäyspuun ideana on tehdä järjestelmällinen ja hierarkinen esitys uhasta. Esimerkki hyökkäyspuusta on *kuvassa 6.*
[15]



Kuva 6: Esimerkki hyökkäyspuusta [15]

Hyökkäyspuussa pyritään tunnistamaan onnistuneen hyökkäyksen tavoite. Tavoitetta pyritään sitten jakamaan osatavoitteisiin, jotka taas edelleen osatavoitteisiin. Puun avulla nähdään helposti, mitä hyökkääjän tulee saavuttaa, jotta hän onnistuu päämäärässään, mutta puusta nähdään myös, mikä kohta on helpoimmin sovelluksessa korjattavissa. Uhan riskiä vähentävien osaratkaisujen keksiminen on helpompaa, kun pystytään havainnoimaan hyökkääjän osatavoitteita.
[15]

Hyökkäyskaavat ovat yleisiä esityksiä hyökkäyksistä. Ne määrittelevät hyökkääjän tavoitteen, uhalle vaadittavat ehdot, hyökkäyksen vaiheet, sekä sen lopputuloksen. Kaavat keskittyvät hyökkäykseen vaadittaviin tekniikoihin, kun STRIDE puolestaan keskittyy hyökkääjän tavoitteisiin. Kaava voidaan esittää taulukkomuodossa, mistä esimerkkinä *taulukko 2*.
[15]

Kaava	Koodin injektiohyökkäys
Tavoite	Koodin suorittaminen
Vaaditut ehdot	Huono syötteen tarkistaminen, Hyökkääjän koodilla on tarvittavat oikeudet palvelimella
Hyökkäys-tekniikka	1. Löydetään kohdejärjestelmästä ohjelma, jossa on haavoittuva syötteen tarkistus 2. Tehdään hyökkäyskoodi, joka injektoidaan kohdeohjelmaan 3. Tehdään ohjelma, joka syöttää hyökkäyskoodin kohdeohjelman muistiavaruuteen ja pakottaa pinon ylivuodon avulla hyökkäyskoodin suorituksen
Lopputulos	Hyökkääjän vihamielinen koodi suoritetaan

Taulukko 2: Esimerkki hyökkäyskaavasta [15]

3.8 Uhkien dokumentointi

Uhkien dokumentoinnissa listataan kaikki aikaisemmin esiin nousseet uhat. Tässä vaiheessa ei vielä arvioida uhkien riskejä, vaan kirjataan ne taulukkomaiseen muotoon. Jokaisesta uhasta tulisi mainita seuraavat asiat:

- Uhan kohde
- Riski (arvioidaan vasta seuraavassa vaiheessa)
- Hyökkäyskeino
- Vastatoimenpiteet

Esimerkki:

- Kohde: tietokantakomponentti
- Hyökkäyskeino: SQL-injektio
- Vastatoimenpiteet: Käytä syötteen tarkistamiseen säännönmukaisia lausekkeita (engl. regular expression) ja käytä SQL-parametreja tietokantakyselyssä

[15]

3.9 Uhkien järjestäminen

Uhkien järjestämisessä voidaan käyttää monia keinoja uhkan luokittelemiseksi. Monipuolisen arvion antaa menetelmä nimeltä DREAD, jossa arvioidaan uhkaa viiden kriteerin näkökulmasta. Jokaista kriteeriä arvioidaan asteikolla 1-3 välillä, minkä jälkeen arviot summataan yhteen ja arvioidaan tämän luvun perusteella uhan luokittelua. Jos summa on 12-15, riski luokitellaan korkeaksi. Jos summa on 8-11, riski on keskimääräinen, ja summan ollessa alle 8 luokitellaan riski vähäiseksi. DREAD muodostuu seuraavista arviokriteereistä:

- Vahingon potentiaalisuus (engl. **D**amage potential)
- Toistettavuus (engl. **R**eproducibility)
- Hyödynnettävyys (engl. **E**xploitability)
- Uhan vaikutuksen alla olevat käyttäjät (engl. **A**ffected users)
- Löydettävyys (engl. **D**iscoverability)

Vahingon potentiaalisuus määrittelee, kuinka paljon vahinkoa kyseisellä hyökkäyksellä voidaan saada aikaan. Vahinko voi vaihdella vähäisen tiedon vuotamisesta koko järjestelmän laajuisten oikeuksien saamiseen asti. *Toistettavuus* arvioi voiko hyökkäyksen suorittaa joka kerta, vai onko sitä hankala toistaa, vaikka turvallisuusaukko olisi tiedossa. *Hyödynnettävyys* kuvaa hyökkääjältä vaadittavia taitoja, ts. voiko hyökkäyksen tehdä jo aloitteleva ohjelmoija vai vaatiiko se laajaa kokemusta jokaisella suorituskerralla. *Uhan vaikutuksen alla olevilla käyttäjillä* tarkoitetaan niiden käyttäjien määrää, joita kyseinen uhka koskee. Määrään vaikuttaa se, onko kyseisen uhkan osatekijänä asetus, joka on oletusarvoisesti päällä, vai vaatiiko uhka hyvin erikoisen konfiguraation toteutuakseen. *Löydettävyys* kuvaa kuinka helposti kyseinen uhka löytyy sovelluksesta, ts. onko uhka sovelluksen eniten käytetyssä vai erittäin vähän käytetyssä osassa.

([6], s. 93-95; [15])

3.10 Yhteenveto

Kappaleessa käsiteltiin menetelmää, jonka avulla voidaan mallintaa sovellukseen liittyviä uhkia. Menetelmän tavoitteena on nostaa esiin uhat siten, että niiden vaarallisuuskin olisi arvioitu. Tämän tiedon avulla uhkia voidaan minimoida ja poistaa mahdollisimman tehokkaasti ”vaarallisimmat ensin” –

periaatteella. Menetelmästä saatuja tietoja voivat käyttää sovelluskehittäjien lisäksi myös suunnittelijat entistä turvallisempien sovellusarkkitehtuurien kehittämiseen, sekä testaajat, jotka voivat testata sovelluksen haavoittuvaisuuksia menetelmästä saatujen tietojen pohjalta.
[15]

Pelkästään uhkien arvioiminen ja tiedostaminen ei kuitenkaan riitä, vaan uhat pitää pystyä poistamaan konkreettisin toimin. Kappaleessa 4 käydään läpi muutamien yleisimpien uhkien poistamismenetelmiä.

4 Tietoturvaongelmia aiheuttavat ohjelmointivirheet

4.1 Johdanto

Tässä luvussa esitellään yleisimpiä ohjelmointivirheitä ja suunnitteluvirheitä. Jokaisesta tietoturvaongelmasta on kuvaus, yksinkertainen esimerkki ja parannusehdotuksia. Kuvauksessa kerrotaan minkälaisesta taustasta tai mistä syystä johtuen kyseinen virhe johtaa tietoturvaongelmaan. Esimerkki havainnollistaa virheen ja tietoturvaongelman yhteyden. Esimerkissä on esitetty myös hyökkäys virhettä vastaan. Aluksi käsitellään ohjelmointikieliriippuvaista tunnettua tietoturvaongelmaa nimeltä puskurinylivuodot. Puskurinylivuodoista esitellään pinon ja keon ylivuodot. Tämän jälkeinen loppuluku käsittelee web-ohjelmointiin liittyviä tietoturvaongelmia. Web-ohjelmointiosan alussa on myös lyhyt opastus web-ohjelmoinnin siirtoprotokollaan http:hen, jonka jälkeen käydään läpi erityisesti web-ohjelmointia koskevat yleisimmät tietoturvaongelmat.

4.2 Puskuriylivuodot

4.2.1 Kuvaus puskurin ylivuodoista

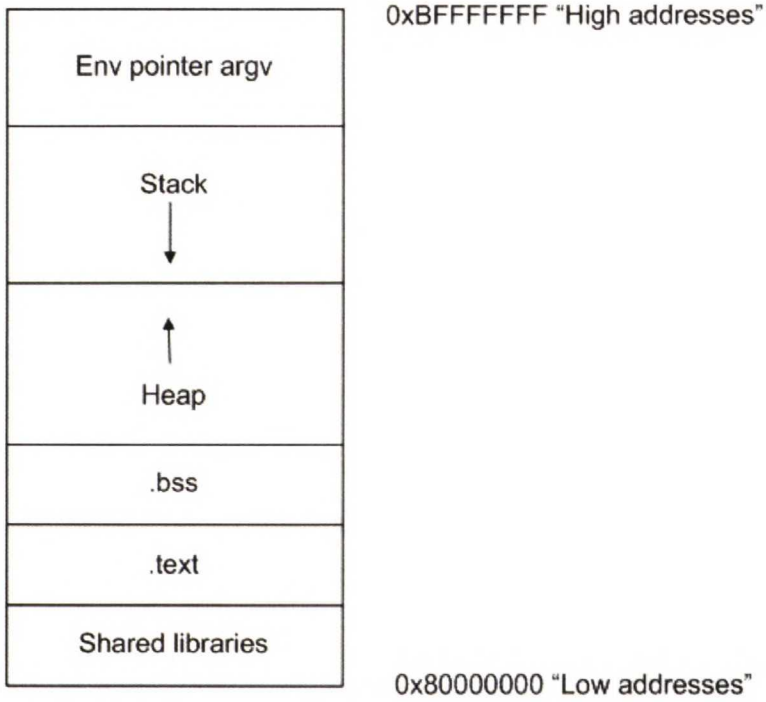
Puskurin ylivuoto tapahtuu, kun käsitellään sovelluksessa dataa puskurissa, jolle varattu koko on pienempi kuin siihen sijoitettava data. Tämä voi johtaa ohjelman kaatumiseen, väärään toimintaan tai hakkerin laatiman koodin suoritukseen. Jos ylivuoto on tapahtunut hakkerin toimesta, ja hänen laatimansa koodi suoritetaan, se suoritetaan samoilla oikeuksilla kuin kyseisellä sovelluksella oli suoritushetkellä. Jos puskurin ylivuoto on tapahtunut käyttöjärjestelmän pääkäyttäjän oikeuksilla, voi hakkeri suorittaa omalla koodillaan samoja toimenpiteitä kuin pääkäyttäjä, mikä käytännössä tarkoittaa kaikkiin tiedostoihin esteetöntä luku- ja kirjoitusoikeutta. Erittäin yleiseksi ja vaaralliseksi puskurinylivuodon tekee se, että se koskee erityisesti C-ohjelmointikieltä, jolla on tehty lähes kaikki modernit käyttöjärjestelmät. Lisäksi C-ohjelmointikielellä on tehty sen yli 30-vuotisen historian aikana monia muitakin suuria sovelluksia, kuten tietokantapalvelimia ja muita palvelinsovelluksia, jotka ovat erityisen kiinnostavia hakkereiden näkökulmasta. Puskurin ylivuoto tapahtuu erityisesti C-ohjelmointikielessä helposti, koska siinä ei automaattisesti tarkisteta taulukoiden ja merkkijonojen rajoja. Ohjelmoija voi siis varata merkkijonolle

tilaa esimerkiksi 20 merkin verran, vaikka todellisuudessa merkkijonopuskuriin kirjoitetaan enemmän merkkejä. Tällöin merkkijono ylikirjoittaa muistissa sille varatun tilan jälkeisiä tavuja. Tämä johtaa helposti vaarallisiin tilanteisiin, kuten kappaleet 4.2.2 ja 4.2.3 osoittavat.
([1], s. 341-344; [2], s. 8, 12; [11], s. 303; [16])

4.2.2 Pinon ylivuoto

4.2.2.1 Kuvaus pinon ylivuodosta

Pino (engl. stack) on rakenteeltaan LIFO-tyyppinen (Last In First Out), eli siihen viimeiseksi talletettu tieto on pinon päällimmäisenä. Pino on ihanteellinen rakenne väliaikaisen tiedon talletuspaikkana. Pinoon talletetaan paikalliset muuttujat, funktiokutsut ja muut tiedot, joita tarvitaan funktiokutsusta palattaessa. Pino on sijoitettu muistiin siten, että se kasvaa alaspäin. Jos pinoon lisätään jotain, pinon pään muistiosoite pienenee, katso *kuva 7*.
([2], s. 5)



Kuva 7: Muistin rakenne, jossa näkyy pinon sijainti muistissa ([2], s. 6)

Ohjelmoijalle pinon päähän osoittava muistiosoitin näkyy pino-osoittimen rekisterin (engl. extended stack pointer, ESP) avulla. Se osoittaa muistiosoitteeseen, jossa on pinon pää, ts. kohtaan, jossa on viimeksi talletettu arvo. Assem-

bler –kielellä pinoa hallitaan PUSH- ja POP-komennoilla, joista PUSH työntää pinoon arvon ja POP poistaa sieltä arvon. Kun PUSH-komento suoritetaan, pino-osoittimen arvoa vähennetään esim. neljällä, kun käytetään 32-bittistä muuttujaa (engl. double word, dword), ja tähän uuteen osoitteeseen sijoitetaan haluttu arvo. Vastaavasti POP-komennossa ensin luetaan arvo pino-osoittimen osoittamasta paikasta, minkä jälkeen pino-osoittimen arvoa kasvatetaan. On huomioitava myös se, ettei muistissa olevia arvoja tyhjennetä tai tuhota, joten pinossa olevat tiedot jäävät muistiin, kunnes ne seuraavan kerran ylikirjoitetaan.
([2], s. 13-14)

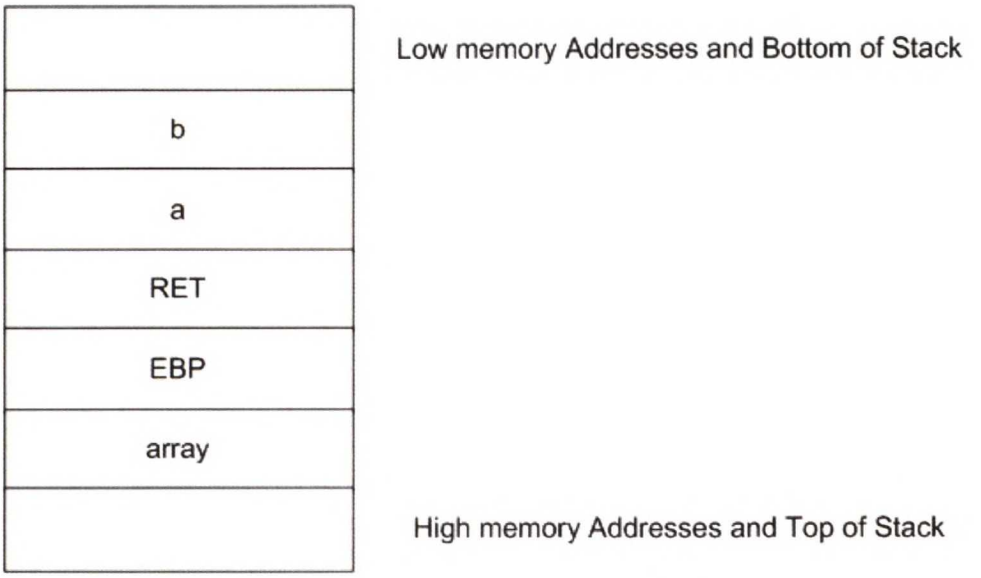
Pinoa käytetään erityisesti funktiokutsujen yhteydessä. Pino mahdollistaa sen, että funktio toimii itsenäisesti irrallaan muusta ohjelmasta, jolloin ohjelman logiikkaa voidaan jakaa pienempiin ja helpommin käsiteltäviin osiin. Kun funktiota kutsutaan, ohjelman suoritus sen funktion sisälle, ja kun funktio on suoritettu, ohjelma palaa samaan pisteeseen, josta kyseistä funktiota oli alun perin kutsuttu. Esimerkki yksinkertaisesta funktion kutsusta:
([2], s. 14-15)

```
1   void function(int a, int b )
2   {
3           int array[5];
4   }
5   main()
6   {
7           // Code before function
8           function( 1, 2 );
9           // Code after function
10  }
```

Koodi 1: Yksinkertainen funktion kutsu ([2], s. 15)

Tässä esimerkissä ohjelman suoritus etenee main-funktiossa, kunnes tullaan funktiokutsun kohdalle riville 8. Funktiokutsu suoritetaan siten, että ensin sijoitetaan kutsun parametrit pinoon oikealta vasemmalle lukien. Tämän jälkeen pinoon sijoitetaan funktion palaamisen jälkeinen osoite, joka on käskyosoitti-

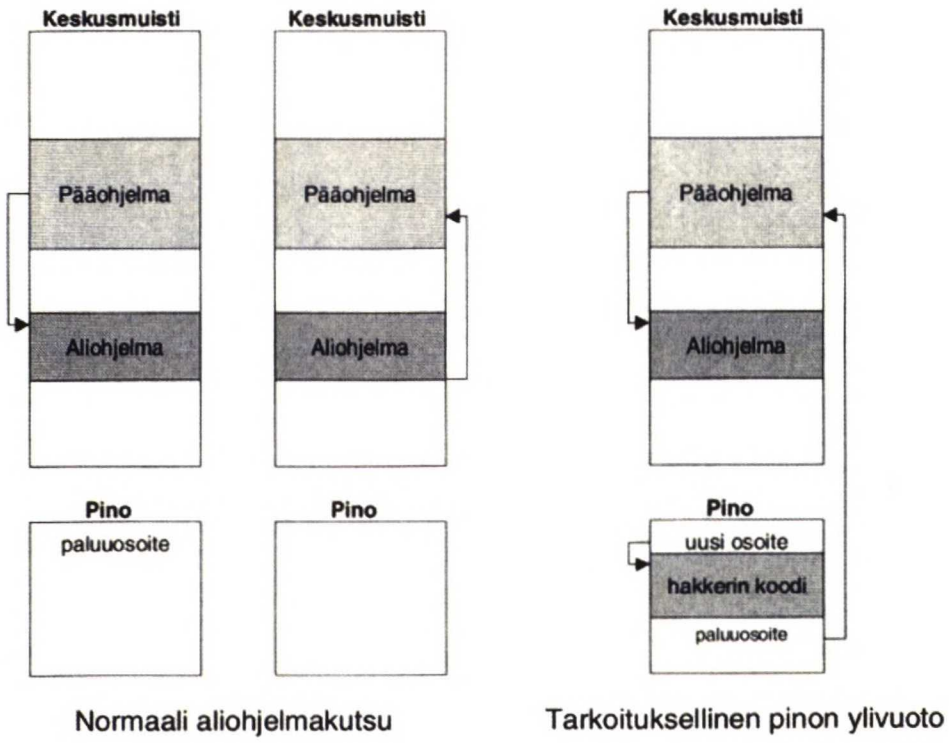
messa (engl. extended instruction pointer, EIP) tällä hetkellä oleva osoite. Tästä osoitteesta ohjelman suoritus jatkuu, kun funktio on suoritettu. Seuraavaksi suoritetaan alustavat toimet ennen funktion käsittelyä (engl. procedure prolog), jotta funktiota voidaan puhtaasti kutsua. Tämä tarkoittaa, että kutsujan perusosoittimen (engl. extended base pointer, EBP) arvo talletetaan pinoon (engl. saved frame pointer, SFP), josta se myöhemmin voidaan palauttaa alkuperäiseen arvoonsa. Funktion paikallisille muuttujille varataan tila pinosta, minkä jälkeen funktiossa oleva koodi suoritetaan. Kyseisessä funktiossa on array-niminen puskuri (ts. taulukko), jossa on tilaa 5 alkiolle (array[0]-array[4]). C-ohjelmointikielessä puskurien rajoille ei ole mitään tarkistuksia, joten array-puskuriin voisi kopioida yli 5 alkiota dataa, vaikka siihen ei mahtuisikaan. *Kuvassa 8* näkyy pinon tila tässä vaiheessa.
([2], s. 15-16; [3], s. 18-21; [17], s. 98)



Kuva 8: Pinon tila, kun koodin 1 mukainen ohjelma on suorituksessa function – funktiossa ([2], s. 16)

Kuvassa 8 näkyy se, että paikallisille muuttujille varattu tila on aivan paluuosoittimen viereisissä muistipaikoissa, joten jos funktiossa käsitellään array-puskuria huolimattomasti, seuraa siitä helposti puskurin ylivuoto. Tällöin hyökkääjä voisi sijoittaa paluuosoitteen kohdalle haluamansa osoitteen, joka

sitten siirtäisi ohjelman suorituksen hänen haluamaansa paikkaan, kuten *kuvassa 9* näkyy.



Kuva 9: Puskurin vuotaessa yli hyökkääjän koodi saattaa päästä pinoon ja suoritettavaksi ([11], s. 303)

Seuraavassa kappaleessa käydään läpi lyhyt esimerkki, jossa näkyy pinon ylivuoto ja sen hyväksikäyttö.

4.2.2.2 Esimerkki pinon ylivuodosta

Esimerkkiohjelma on hyvin yksinkertainen konsoliohjelma. Käynnistymisen jälkeen se tulostaa käyttäjälle tekstin "Enter your name: ", minkä jälkeen se jää odottamaan käyttäjän syötettä. Kun käyttäjä on syöttänyt nimensä ja painanut enter-näppäintä, ohjelma tulostaa "Hi Name!". Ohjelman lähdekoodi kokonaisuudessaan on listattu *koodi 2:ssa*.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void myFunction()
5   {
6         char buf[ 30 ];
7         printf( "Enter your name: " );
8         gets( buf );
9         printf( "Hi %s!\n", buf );
10  }
11
12  void myOtherFunction()
13  {
14        printf( "In Secret function!\n\n" );
15        exit( 0 );
16  }
17
18  int main(int argc, char* argv[])
19  {
20        myFunction();
21        return 0;
22  }
```

Koodi 2: Puskurinylivuoto esimerkkiohjelman lähdekoodi
([2], s. 18; [6], s. 129-130)

Lähdekoodista voi havaita funktion ”myOtherFunction” rivillä 12, jota ei koskaan kutsuta ohjelmasta käsin, mutta jota voidaan kutsua puskurin ylivuodon avulla. Ohjelma ei näytä ulospäin mitenkään erikoiselta, eikä siinä olevaa virhettä moni havaitse. Ohjelmassa piilee kuitenkin vakava tietoturva-aukko. Virheen havaitsee, kun antaa ohjelmalle syötteen, joka on pidempi kuin rivillä 5 määritellyn puskurin suurin sallittu koko, 30 merkkiä. Virheen seurauksena ohjelma kaatuu, kuten *kuvista 10* ja *11* nähdään.
([2], s. 18-19)

Kuva 10: Liian pitkä syöte on kaatanut ohjelman windowsissa

Vastaava ohjelma linuxissa tuotti *kuvan 11* mukaisen virheilmoituksen.

```
$ ./StackOverrun
Enter your name: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Hi aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!
Segmentation fault (core dumped)
$
```

Kuva 11: Liian pitkä syöte on kaatanut ohjelman linuxissa

Käyttäjän syöte - 40 kpl a-kirjaimia - on ylikirjoittanut muistista funktion paluuosoitteen. Tämän voi havaita windowsin virheilmoituksesta, jossa sanotaan ohjelman yrittäneen käsitellä muistiosoitetta 0x61616161, missä 0x61 on heksadesimaaliluku, joka vastaa ascii-merkistön a-kirjainta. Vastaavan saa selville linuxissa debuggerilla, kuten *kuvasta 12* nähdään.

```
(gdb) info registers
eax            0x37       55
ecx            0x0        0
edx            0x37       55
ebx            0x40244e9c       1076121244
esp            0xbffffd00       0xbffffd00
ebp            0x61616161       0x61616161
esi            0x40246170       1076126064
edi            0xbffffd20       -1073742560
eip            0x61616161       0x61616161
eflags         0x10282    66178
cs             0x73       115
ss             0x7b       123
ds             0x7b       123
es             0x7b       123
fs             0x0        0
gs             0x33       51
```

Kuva 12: Ohjelman kaatuessa EIP:n arvo oli a-kirjaimien ylikirjoittama

Puskurista ylivuotaneet merkit ovat siis molemmissa tapauksissa ylikirjoittaneet pinossa olevan paluuosoitteen, mikä on tässä tapauksessa johtanut ohjelman kaatumiseen. Toisenlaisella syötteellä ohjelmaan olisi saatu syötettyä omaa ohjelmakoodia, jota olisi ajettu ohjelman käytössä olevilla oikeuksilla. Äärimmäisen vakavaksi kyseinen virhe tulee siis tapauksissa, joissa ohjelmaa ajetaan pääkäyttäjän tai järjestelmän oikeuksilla. Jos ohjelmaa olisi käytetty ”oikein”, eli sille ei olisi annettu ylipitkää syötettä, ohjelma voisi olla pitkäänkin käytössä kenenkään huomaamatta ongelmaa. Jos käyttötilanteessa tulisi vastaan ylipitkä syöte aiheuttaen ohjelmaan kaatumisen, olisi virheen etsiminen ja löytyminen ollut paljon todennäköisempää.

Puskurin ylivuotoa hyödyntäen voidaan tehdä hyökkäys (engl. exploit), jolla pystytään osoittamaan tietoturvaongelman vaarallisuus. *Koodiesimerkissä 3* on listaus ohjelmasta, joka kutsuu puskuriylivuotohyökkäyksen avulla ohjelmassa olevaa ”myOtherFunc”-funktiota.

```
1   #include <stdio.h>
2   #define OVERFLOWSIZE  40        // linux: 40
3   #define JUMPADDR      0x401030 // linux: 0x080483f6
4
5   main()
6   {
7     int i = 0;
8     char stuffing[ OVERFLOWSIZE ];
9     for( i = 0; i <= OVERFLOWSIZE-8; i++ )
10    {
11      stuffing[ i ] = 'a';
12    }
13    *( long * ) &stuffing[OVERFLOWSIZE-4] = JUMPADDR;
14    puts( stuffing );
15  }
```

Koodi 3: Yksinkertainen ohjelma, jolla voidaan suorittaa puskuri ylivuoto-hyökkäys ([2], s. 21)

Lähdekoodissa on kaksi muuttujaa, joiden arvoja muuttamalla hyökkäys saadaan toimimaan eri järjestelmissä. Oletusarvoina ovat Windows-järjestelmään sopivat arvot ja kommenteissa linuxiin sopivat arvot. Nämä arvot vaihtelevat kääntäjästä ja ympäristöstä riippuen. Tarvittavat luvut saadaan esiin ohjelman binäärin assembler-koodista (engl. reverse engineering; disassembling), josta nähdään kuinka paljon pinosta on itse asiassa varattu tilaa. *Kuvissa 13* ja *14* näkyy, kun hyökkäysohjelmaa käytetään sovellustamme vastaan.

Kuva 13: Pinon ylivuotohyökkäyksen käyttö windowsissa

```
$ ./StackOverrunExploit | ./StackOverrun
Enter your name: Hi aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaÙÖ@↳Þ@ d@

                                                        ýÿ¿ø×@!
In Secret function!

$
```

Kuva 14: Pinonylivuotohyökkäyksen käyttö linuxsissa

Kuvista nähdään, kuinka hyökkäysohjelma on onnistuneesti kutsunut piilossa olevaa funktiota. Hyökkäys on tietysti hyvin yksinkertainen, mutta samalla periaatteella toimivat monimutkaisemmatkin hyökkäykset. Usein hyökkäyksessä pyritään käynnistämään komentotulkki (engl. shell), jonka avulla hyökkääjä pystyy suorittamaan haluamiaan komentoja. On huomattava, että hyökkääjä saa käyttöönsä ne oikeudet, jotka sovelluksella on käytössä, joten etenkin palvelinohjelmissa olevat haavoittuvuudet ovat erittäin vaarallisia. Hyökkäys on myös täysin toistettavissa, kun se kerran on saatu toimimaan. Näiden ominaisuuksien takia monet madot hyödyntävät juuri pinon ylivuodon aiheuttamia tietoturva-aukkoja levitessään.

4.2.2.3 Parannusehdotuksia pinon ylivuotoon

Pinon ylivuoto on niin vaarallinen ja yleinen ongelma, että sitä vastaan on kehitetty puolustautumiskeino kääntäjiin. Kääntäjä lisää koodiin tarkistuksia, jolla pystytään havaitsemaan pinon ylivuodot ajonaikaisesti. Toiset kääntäjät ilmoittavat taas käännösvaiheessa, että kyseisessä kohdassa on pinon ylivuotomahdollisuus, ja se pitäisi välittömästi korjata. Laitetasollekin on rakennettu suojauksia, joilla pyritään estämään ohjelmakoodin ajaminen pinossa (engl. no execute stack). Käyttöjärjestelmiinkin on parannettu tarkistuksia, joilla saataisiin estettyä pinon ylivuotovirheet. Näillä keinoilla ei kuitenkaan pystytä täysin varmasti estämään mahdollisia hyökkäyksiä. Puolustuskeinoja vastaan on myös kehitetty ohittamiskeinoja, joiden avulla mahdollinen suojaus ei enää toimikaan, kuten alun perin on suunniteltu. Ainoaksi varmasti toimivaksi puolustuskeinoksi pinon ylivuotoja vastaan jää siis puhtaan koodin kirjoittaminen. ([1], s. 184-185; [2], s. 29-33; [6], 138-139, 155-156, 167-170)

4.2.3 Keon ylivuoto

4.2.3.1 Kuvaus keon ylivuodosta

Pinoa käytettiin paikallisten muuttujien tallentamiseen, mutta globaaleiden muuttujien tallentamiseen pino käy liian pieneksi. Globaalien muuttujien tarvitseman tilan määrää ei vielä välttämättä edes tiedetä ohjelman käännösvaiheessa, joten muistinvaraus tehdään tällöin ajonaikaisesti järjestelmäkutsujen avulla. *Kuvassa 7* näkyi ".bss" -osa, johon on sijoitettu alustamattomat globaa-

lit muuttujat, ja alustetut muuttujat ovat vastaavasti ".data"-osuudessa. Keoiksi kutsutaan niitä muistialueita, jotka on erikseen varattu dynaamisella muistinvarauksella. Keon ylivuodoista puhuttaessa kuitenkin yleisesti tarkoitetaan myös ".bss" ja ".data" –osuuksia, koska nekin ovat haavoittuvaisia näille hyökkäyksille. Keon ylivuotohyökkäykset, kuten pinon ylivuotohyökkäyksetkin, ovat seurausta syötteestä, jolle ei ole varattu riittävästi tilaa. Tämän seurauksena ylivuotava osa voi täyttää muistissa olevia toisia puskureita ja niitä varten käytettäviä metatietoja. Muistinvarauksia ja vapautuksia hoitavat funktiot (esim. malloc ja free) tarvitsevat metatietoja, joiden avulla ne voivat hallita muistia. Ne säilyttävät niitä omissa globaaleissa muuttujissaan sekä lisäksi varattujen muistilohkojen ennen tai/ja jälkeen. Aivan kuten pinon ylivuotohyökkäyksessä, myös tässä pyritään ylivuotavalla osalla vaikuttamaan siihen muistin osaan, joka on ylivuotaneen puskurin jälkeen. Ylivuotavalla osalla voidaan huijata muistinhallintafunktioita tekemään erilaisia toimintoja, kuten ylikirjoittaa tiedostonimien päälle tai kirjoittaa sellaiseen muistipaikkaan, jota hyökkääjä voi hallita.

([2], s. 84-88; [3], s. 41-44; [6], s. 138-144; [18])

4.2.3.2 Esimerkki keon ylivuodosta

Esimerkkisovellus kuvaa yksinkertaista sisäänkirjautumista, jossa käyttäjä antaa komentorivillä käyttäjätunnuksen ja sovellus tarkistaa käyttäjän oikeudet. Tämän jälkeen sovellus sulkee itsensä, jos käyttäjällä ei ole ylläpitäjän oikeuksia. Ylläpitäjä saisi jatkaa sovelluksen käyttämistä ylläpitotarkoituksiinsa. *Kuvissa 15* ja *16* näkyy sovelluksen tulostus, kun käyttäjä ei ole läpäissyt kirjautumista.

Kuva 15: Sisäänkirjautuminen epäonnistui windows-järjestelmässä

```
$ ./HeapOverrun Me
Access denied!
$
```

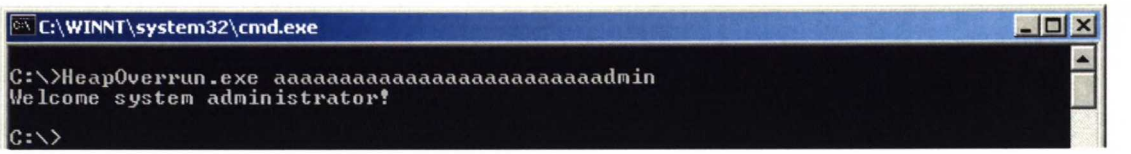
Kuva 16: Sisäänkirjautuminen epäonnistui linux-järjestelmässä

Sovelluksen koodi on kokonaisuudessaan liitteessä 1, mutta sen toiminnallisesti tärkein osuus on seuraavanlainen:

```
1   char *login;
2   char *username;
3
4   login = (char*) malloc(10);
5   username = (char*) malloc(10);
6
7   // Set current username for anonymous:
8   strcpy( username, "anonymous" );
9
10  // Get users loginname:
11  strcpy( login, argv[1] );
12
13  // <Check login information>
14
15  if ( strcmp( username, "admin" ) == 0 )
16        printf("Welcome system administrator!\n");
17  else
18        printf("Access denied!\n");
```

Koodi 4: Sisäänkirjautuminen, jossa on keon ylivuotohaavoittuvaisuus

Koodin alussa riveillä 4 ja 5 varataan molemmille muuttujille tilaa keosta 10-merkin verran. Tämän jälkeen alustetaan käyttäjätunnus anonyymiksi rivillä 8. Rivin 13 kohdalle kuuluu sovelluksen logiikka, joka etsisi kyseiselle käyttäjälle oikean käyttäjätunnuksen tässä sovelluksessa. Rivillä 15 tarkastetaan, onko käyttäjätunnus ylläpitäjä, jolloin hän pääsisi käyttämään ylläpitäjälle tarkoitettuja toimintoja. Jos kuitenkin käyttäjä antaa syötteeksi riittävän pitkän merkkijonon, ylikirjoittaa se muistissa myös sovelluksen sisäisen käyttäjätunnuksen. Tästä seuraa *kuvien 17* ja *18* mukainen tilanne:



Kuva 17: Keon ylivuotohyökkäys windows-järjestelmässä

```
$ ./HeapOverrun aaaaaaaaaaaaaaaaadmin
Welcome system administrator!
Segmentation fault (core dumped)
$
```

Kuva 18: Keon ylivuotohyökkäys linux-järjestelmässä

Molemmissa järjestelmissä ylivuoto on aiheuttanut virheen sovelluslogiikassa. Linux:issa sovellus kaatui sovelluksen lopuksi, kun varattuja muisteja vapautettiin (funktio free). Se ei kuitenkaan olisi vaikuttanut hyökkäykseen, koska hyökkääjä olisi voinut tehdä toimintonsa rauhassa ja vasta sitten poistunut sovelluksesta, minkä jälkeen se olisi kaatunut.

4.2.3.3 Parannuskeinoja keon ylivuotoon

Keon ylivuoto on ohjelmoijan kannalta pinon ylivuotoa ikävämpi, koska ei ole pystytty rakentamaan automaattisia työkaluja, jotka etsisivät mahdollisia keon ylivuotoja sovelluksesta. Näin ollen kääntäjiinkään ei ole pystytty luomaan apukeinoa, joka ilmoittaisi mahdollisista virheistä tai lisäisi koodia, jolla virhetilanne saataisiin kiinni sen sattuessa. Virheiden etsiminen on myös hankalaa, koska virhe saattaa tulla ilmi vasta monen asian yhdistelmänä. Keon ylivuotojen monimutkaisuuden takia monet ohjelmoijat eivät myöskään usko, että niitä voidaan hyödyntää, vaikka monet julki tulleet tietoturvaongelmat ja levinneet madot kertovat aivan toista. Keon ylivuodonkin kohdalla ainoa todellinen suojautumiskeino on parempi koodin laatu, ts. huolellinen syötteiden tarkistus.

([2], s. 85-87; [6], s. 138-139)

4.3 Yleisimmät web-hyökkäykset

4.3.1 Johdanto

Internetin suosio perustuu hyvin pitkälle siihen, että palvelua pystytään käyttämään pitkienkin välimatkojen päästä ilman mitään huomattavaa nopeuseroa. Laaja käyttäjäkunta taas tuo väistämättä myös vihamieliset käyttäjät eri sovellusten piiriin. Lisäksi monet web-sovellukset ovat yrityksien toiminnan kannalta elintärkeitä, kuten erilaiset verkkokaupat sekä pankki- ja vakuutuspalvelut. Tällaisissa sovelluksissa tietoturvan merkitys on äärimmäisen suuri, jotta luottamus asiakkaan ja yrityksen välillä säilyy. Kriittiset sovellukset houkuttelevat vielä tavallista enemmän vihamielisiä hyökkääjiä ympärilleen.

Tässä luvussa käydään läpi yleisimmät uhat, jotka kohdistuvat web-sovelluksiin. Ensin käsitellään lyhyesti HTTP:tä (kappale 4.3.2) teknisen taustan selvittämiseksi, minkä jälkeen käydään läpi web-lomakkeiden ja keksien arvojen väärentämiseen liittyviä uhkia, selaimen ohjelmointiin liittyviä uhkia, SQL injektio-uhkaa, palvelunestohyökkäyksen uhkaa ja verkkoliikenteen kuuntelusta seuraavia uhkia.

4.3.2 HTTP lyhyesti

Hypertext Transfer Protocol määrittelee sovellustason protokollan hajautettuun järjestelmään. Sen toiminta perustuu pyyntö-vastaus (engl. request-response) – menetelmään, jossa asiakas (engl. client) pyytää palvelimelta (engl. server) tiedostoa, ja palvelin vastaa asiakkaan pyyntöön palauttamalla tälle tiedoston. Asiakas pystyy pyynnössään välittämään palvelimelle metatietoja, jotka auttavat palvelinta tuottamaan sivun asiakkaalle. Esimerkki sivupyynnöstä on *koodiesimerkissä 5*.

([19], s. 7,12)

```
1  GET /index.html HTTP/1.1
2  Accept: image/gif, image/jpeg, */*
3  Accept-Language: fi
4  Accept-Encoding: gzip, deflate
5  User-Agent: Mozilla
6  Host: localhost
7  Connection: Keep-Alive
8  Cookie: myAppCookie=my secret cookie
9
```

Koodi 5: Esimerkki asiakkaan lähettämästä sivupyynnöstä palvelimelle

Esimerkissä näkyy ensimmäisellä rivillä sivu, jonka asiakas haluaa (index.html), käytettävä protokollaversio (1.1), sekä menetelmä, jolla asiakas haluaa sivua hakea (GET). Pyynnön mukana on myös rivillä 8 eväste, ts. keksi, (engl. cookie), jonka avulla palvelin voi tunnistaa asiakkaan. Tämä mahdollistaa käyttäjäkohtaisen sivun muodostamisen. Sivupyyntö loppuu esimerkissä kahteen rivinvaihtoon (rivi 8 ja 9), joiden avulla erotetaan sovellukselle lähetetty viestiosa (engl. body) otsakeosasta (engl. header). Esimerkissä ei lähetetty palvelimelle mitään viestiosassa, joten se on tyhjä. Asiakas voi välittää sivupyynnön palvelimelle myös kyselymuodossa (engl. query), jolloin parametrit tulevat halutun dokumentin nimen perään "?" ja "&"-merkeillä erotettuina:

```
GET /index.html?param1=value1&param2=value2 HTTP/1.1
...
```

Koodi 6: Asiakkaan sivupyyntö, jossa parametrit ovat kyselymuodossa

Palvelimen vastaus asiakkaan *koodin 5* mukaisen sivupyyntöön näkyy *koodissa 7.*

```
1   HTTP/1.1 200 OK
2   Server: My Server Software
3   Date: Mon, 12 Jul 2004 07:34:49 GMT
4   Set-Cookie: id=123; path=/
5   Content-Type: text/html
6   Accept-Ranges: bytes
7   Last-Modified: Mon, 12 Jul 2004 07:33:16 GMT
8   ETag: "90ed87fe267c41:81c"
9   Content-Length: 55
10
11  <html>
12  <body>
13  Yksinkertainen sivu
14  </body>
15  </html>
```

Koodi 7: Palvelimen vastaus asiakkaan lähettämään kyselyyn

Palvelimen vastauksen ensimmäisellä rivillä näkyy vastauksessa käytetty protokollaversio ja vastauksen tila, joka on tässä esimerkissä 200 ja ok. Rivillä 4 palvelin pyytää asiakasta tallentamaan itselleen evästeen nimeltä ”id” arvolla ”123”. Seuraavan kerran asiakkaan tehdessä sivupyynnön palvelimelle, hän lisää tämän evästeen omaan kutsuunsa. Palvelimen vastauksen viestiosa on riveillä 11–15. Viestiosan sisältö on Hypertext Markup Language:a (HTML), joka on kuvauskieli sivun esittämiseen asiakkaan selaimessa. Jos palvelimen tuottamalla web-sivulla on lomake, jonka tiedot asiakas haluaa lähettää eteenpäin, voidaan käyttää joko GET-menetelmää, jolloin parametrit tulevat kyselymuotoon, tai POST-menetelmää, jossa parametrit ovat viestiosassa. Esimerkki tästä on *koodi 8*.

```
1   POST /FormFields/myApp HTTP/1.1
2   Accept: image/gif, image/jpeg, */*
3   Referer: http://localhost/FormFields/
4   Accept-Language: fi
5   Content-Type: application/x-www-form-urlencoded
6   User-Agent: Mozilla
7   Host: localhost
8   Content-Length: 30
9   Cookie: myAppCookie=my secret cookie; id=123
10
11  texName=Matti&btnSubmit=Lähetä
```

Koodi 8: Esimerkki asiakkaan sivupyynnöstä, jossa on käytetty POST-menetelmää

Esimerkissä käy ilmi, että pyynnössä on viestiosa, koska otsakeosuudessa kerrotaan rivillä 8 viestiosan pituus. Rivillä 9 näkyy vanhan evästeen lisäksi asiakkaan lähettämä uusi eväste, jonka palvelin oli asiakkaalle asettanut. Viestiosassa rivillä 11 näkyy kaksi parametria, jotka on erotettu "&"-merkillä toisistaan. Tässä muodossa tulevat parametrit, kun käytetään sisällön tyyppinä "x-www-form-urlencoded"-menetelmää, joka on määritelty rivillä 5. GET-menetelmää käytetään vähemmän, koska sen pituus on rajoitettu. Suurin mahdollinen pituus riippuu toteutuksesta, mutta lomakkeen koon kasvaessa raja tulee nopeasti vastaan. Suositeltu käytettävä pituusyläraja on 255 merkkiä ([19], s. 19).

4.3.3 Web-lomakkeiden ja keksien arvojen väärentäminen

4.3.3.1 Kuvaus

Web-lomakkeita käytetään hyvin monenlaisiin tarkoituksiin. Lomakkeilla voidaan syöttää keskustelupalstalle viestejä tai suorittaa ostotapahtumaa verkkokaupassa. Lomake koostuu erityyppisistä kentistä, kuten tekstikentistä, valintalistoista, valintarasteista ja napeista. Valinnoilla voi olla oletusarvoja. Lomakkeella voi olla myös piilokenttiä, joilla on arvo, kuten näkyvilläkin, mutta joita ei lainkaan näytetä selaimessa. Arvojen väärentäminen tarkoittaa sitä, että hyökkääjä syöttää tai muuttaa arvoja sellaisiksi, joihin sovellus ei ole varautunut. Arvojen muuttamisella voidaan saavuttaa sovelluksen toiminnan kannalta

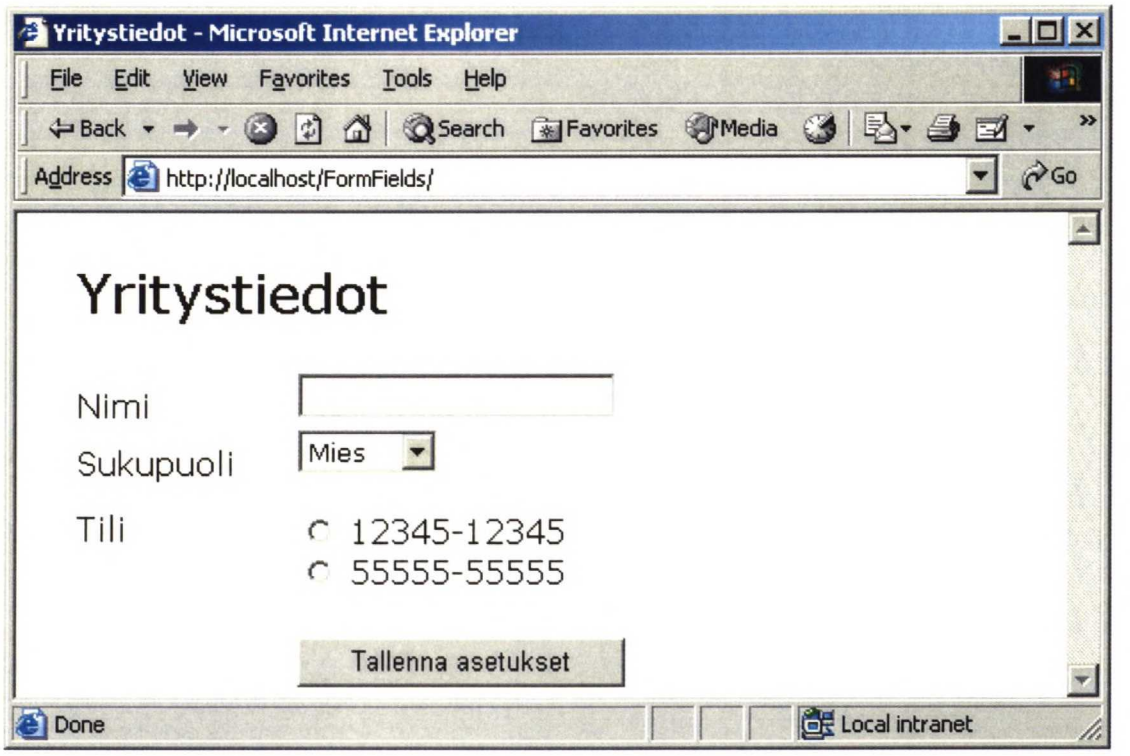
erikoisia ja vaarallisia tilanteita, kuten verkkokaupassa ostoksien hinnan muuttaminen.
([7], s. 93-94)

Keksejä käytetään pääasiallisesti käyttäjän seuraamiseen sivunlatauksien välillä. Niiden avulla käyttäjää voidaan palvella paremmin tiedettäessä mitä hän on aikaisemmin tehnyt. Palvelin antaa sivun latauksen yhteydessä käskyn asiakkaalle keksin tallentamisesta, ja asiakas vastaavasti lähettää palvelimen voimassa olevat keksit takaisin sivupyyntöjen yhteydessä. Monissa sovelluksissa käyttäjäoikeuksien hallinta perustuu myös keksien käyttämiseen. Keksejä voidaan kuitenkin väärentää, jolloin voidaan saada käyttöön toisen henkilön käyttöoikeudet tai kaapattua toisen ihmisen sen hetkinen tila sovelluksessa (engl. session).
([7], s. 80-82, 93-94)

Lomakkeen arvojen muuttamisella ja keksien väärentämisellä on yhteistä se, että molemmissa palvelin luottaa saavansa takaisin asiakkaalta tietoja, jotka palvelin itse on lähettänyt asiakkaalle. Jos asiakas kuitenkin muuttaa saamiaan tietoja, seuraukset voivat olla vakavat, jos palvelimen sovellusta ei ole erikseen suunniteltu sietämään tällaisia hyökkäyksiä.
([7], s. 93-94)

4.3.3.2 Esimerkki lomakkeen väärentämisestä

Esimerkkisovelluksena käytetään tekemääni yksinkertaista sovellusta, jolla hallitaan yritystietoja. Yritystiedoissa on tila henkilön nimelle, sukupuolelle sekä tilille. *Kuvassa 19* näkyy sovellus, kun sivu on ladattu ensimmäistä kertaa.



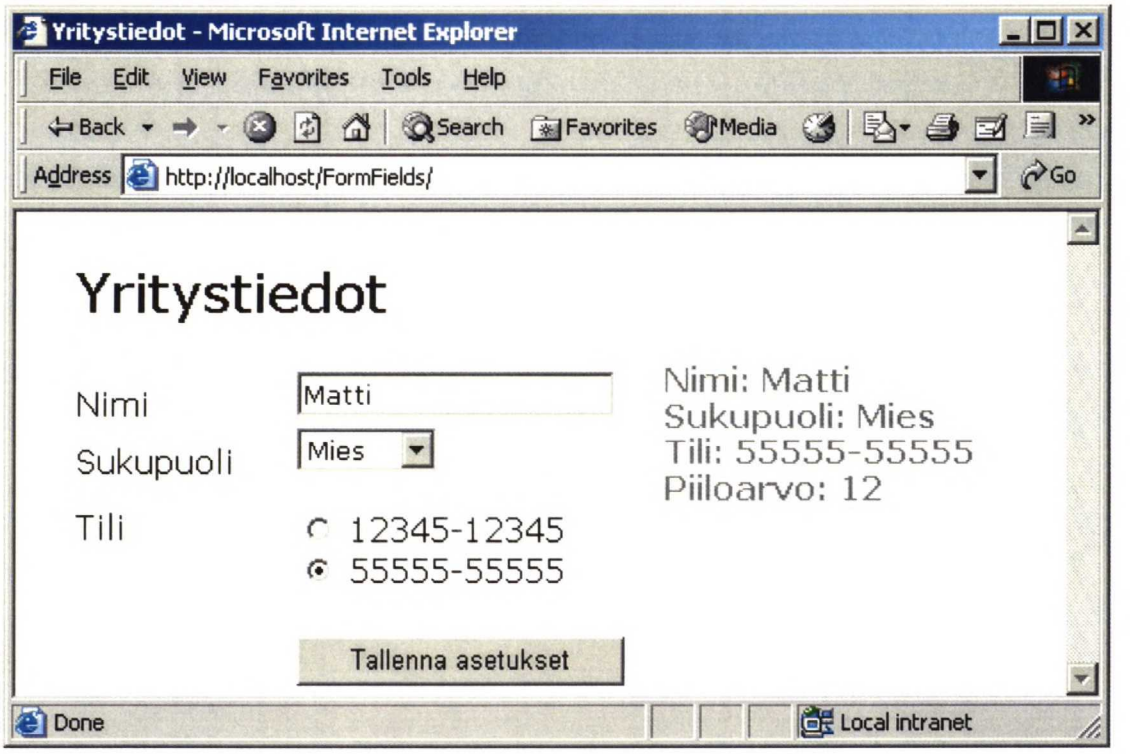
Kuva 19: Esimerkkisovellus, jolla voi muuttaa yritystietoja

Sovelluksessa henkilön nimi on vapaa tekstikenttä (tyyppiä ”text”), sukupuoli on valintalista (tyyppiä ”select”) ja tili on valintanappi (tyyppiä ”radio”). Lisäksi sovelluksessa on käytetty piilokenttää ”code”, jossa on käyttäjältä piilotettu arvo ”12”. Lomakkeen muodostava HTML on *koodissa 9*.

```
1   <FORM method="POST">
2   <INPUT name="texName" type="text">
3   <SELECT name="comGender">
4       <OPTION Value="Mies">Mies</OPTION>
5       <OPTION Value="Nainen">Nainen</OPTION>
6   </SELECT>
7   <INPUT type="radio"
8          name="radAccounts" value="12345-12345">
9   <INPUT type="radio"
10         name="radAccounts" value="55555-55555">
11  <INPUT type="hidden" name="code" value="12">
12  <INPUT type="submit"
12         name="btnSave" value="Tallenna asetukset">
13  </form>
```

Koodi 9: Lyhennetty HTML-koodi esimerkkisovelluksen lomakkeesta

Kun käyttäjä on painanut tallennuspainiketta sivun oikeaan reunaan tulee näkyviin hänen syöttämänsä tiedot. Tästä esimerkkinä on *kuva 20*.



Kuva 20: Käyttäjä on syöttänyt tiedot ja painanut tallennusta

Tallennuspainiketta painettaessa selain lähetti palvelimelle seuraavan sivupyynnön:

```
1   POST /FormFields/myApp HTTP/1.1
2   Accept: image/gif, image/jpeg, */*
3   Referer: http://localhost/FormFields/myApp
4   Content-Type: application/x-www-form-urlencoded
5   User-Agent: Mozilla
6   Host: localhost
7   Content-Length: 87
8
9   texName=Matti&btnSave=Tallenna+asetukset&
    comGender=Mies&radAccounts=55555-55555&code=12
```

Koodi 10: Selaimen lähettämä sivupyyntö palvelimelle tallennusvaiheessa

Palvelimella oleva sovellus on toteutettu siten, että se lukee selaimen lähettämät arvot suoraan ja kaiuttaa ne käyttäjälle, kuten *kuvasta 20* nähtiin. Jos se-

lain lähettäisi väärennettyjä arvoja, palvelin luulisi niiden tulevan suoraan syöttökentistä kaiuttaen näin ollen nekin käyttäjälle. Hyökkäys voidaan toteuttaa käyttäen komponenttia, jolla pystytään jäljittelemään selaimen toimintaa. Nyt kuitenkin muutetaan syöttökenttien arvot sellaisiksi, joita lomake ei tavallisesti sallisi. Esimerkki tästä on *koodissa 11*.

```
1   Browser browser = new Browser( "Mozilla" );
2   browser.GetPage(
        "http://localhost/FormFields/myApp" );
3
4   browser.Forms[0]["texName"] = "Vihamielinen";
5   browser.Forms[0]["comGender"] = "Kissa";
6   browser.Forms[0]["radAccounts"] = "44444-444444";
7   browser.Forms[0]["code"] = "10";
8   browser.Forms[0]["btnSave"] = "Tallenna+asetukset";
9
10  browser.PostPage();
```

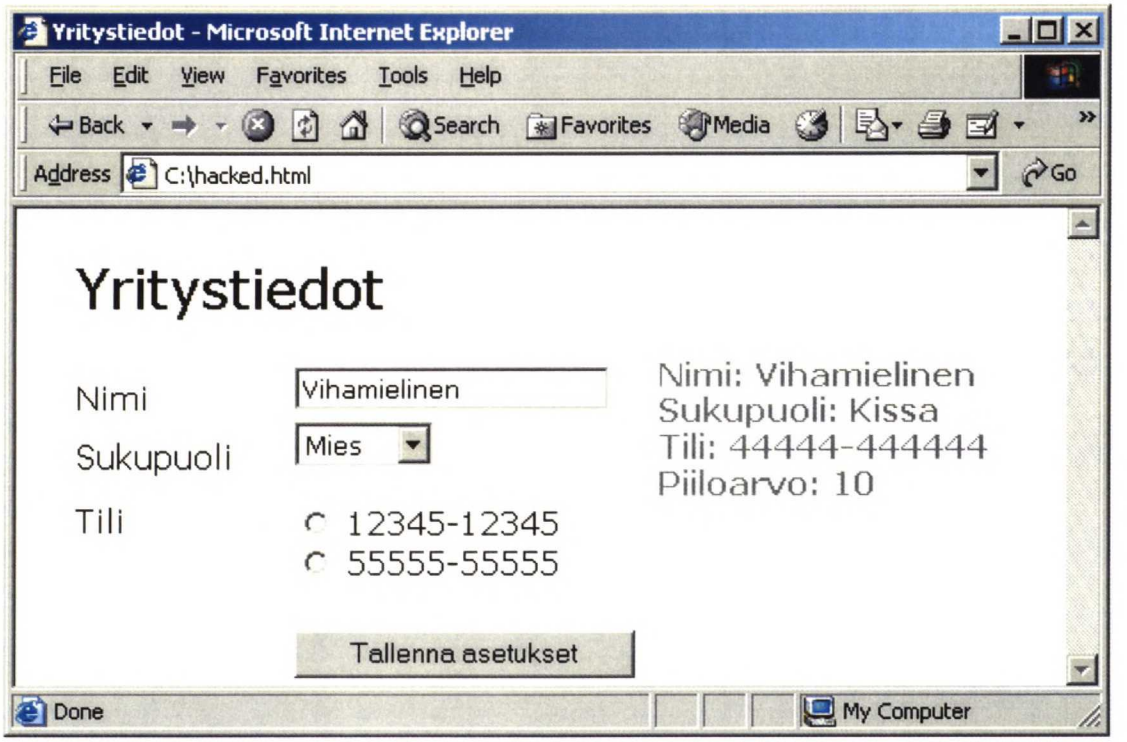
Koodi 11: Web-lomakkeen väärennyshyökkäys

Koodissa näkyy, että sukupuoli kenttään syötetään sana ”Kissa” ja tilin arvoksi ”444444-44444”, joista kumpikaan ei ollut alkuperäisessä lomakkeessa. Lisäksi piilokentän arvoksi muutetaan ”10”. Selain-komponentti generoi samanlaisen sivupyynnön kuin tavallinen selainkin:

```
1   POST /FormFields/myApp HTTP/1.1
2   Content-Type: application/x-www-form-urlencoded
3   User-Agent: Mozilla
4   Host: localhost
5   Content-Length: 96
6
7   texName=Vihamielinen&btnSave=Tallenna+asetukset&
    comGender=Kissa&radAccounts=44444-444444&code=10
```

Koodi 12: Hyökkäyksen muodostama sivupyyntö palvelimelle

Hyökkäyksen seurauksena sovellus palautti *kuvassa 21* esitetyn sivun käyttäjälle:

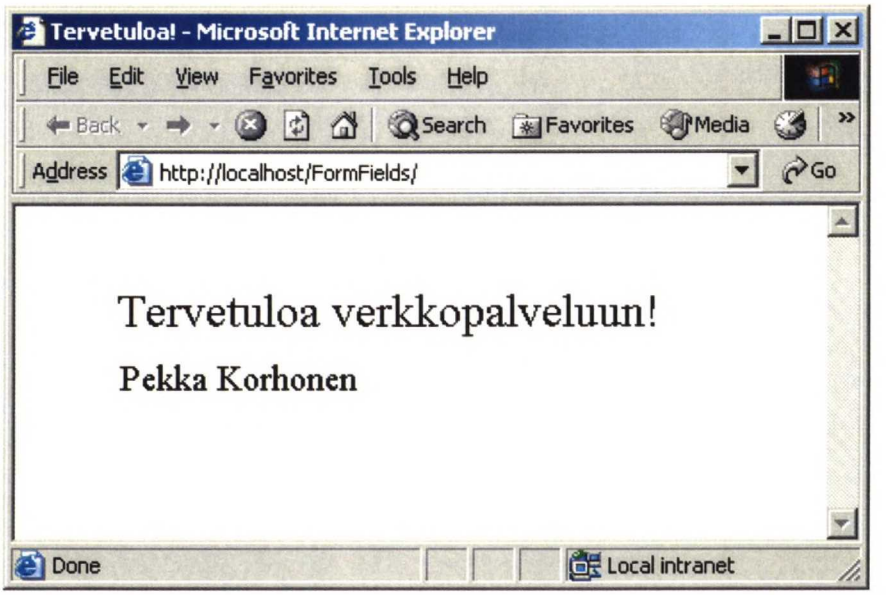


Kuva 21: Sovelluksen palauttama sivu hyökkäyksen seurauksena

Sovelluksen vastauksesta huomaa, että siinä ei tarkasteta käyttäjän syötettä valittavissa olevia syötteitä vasten, vaan arvot luetaan suoraan käyttäjän syötteestä. Piilokentän käytössä täytyy olla tarkkana, koska senkin sisältämää arvoa hyökkääjä pystyy muuttamaan. Tämän takia ei piilokenttiin saa koskaan sisällyttää mitään tärkeää tietoa, joka ei saa muuttua, kuten hintatietoja ([1], s. 622-624). Jos sovellus välittää parametreja kyselymuodossa, ts. url-osoitteen mukana, hyökkääjän on vieläkin helpompi muuttaa kyseisiä arvoja, joten tätä menettelytapaa tulisi mahdollisimman paljon välttää.

4.3.3.3 Esimerkki keksin väärentämisestä

Tekemässäni esimerkkisovelluksessa mallinnetaan tilanne, joka vastaa websovellusten sisäänkirjautumissivun jälkeistä sivua. Käyttäjä toivotetaan tervetulleeksi palveluun hänen nimensä näkyessä sivulla, kuten *kuvassa 22*:



Kuva 22: Sisäänkirjautumisen jälkeinen sivu esimerkkisovelluksessa

Palvelin lisää sisäänkirjautumisvaiheessa asiakkaan selaimeen keksin kutsun yhteydessä:

```
HTTP/1.1 200 OK
...
Set-Cookie: id=2; path=/
...
Content-Length: 914


<HTML>...
```

Koodi 13: Palvelimen palauttama vastaus, jossa näkyy keksin asetus

Tämän jälkeen asiakas lähettää keksin jokaisella sivun latauksella palvelimelle, jotta palvelin tunnistaa asiakkaan oikein. Esimerkki asiakkaan sivupyynnöstä on *koodissa 14*.

```
GET /FormFields/afterLogin HTTP/1.1
Cookie: id=2
...
```

Koodi 14: Asiakkaan sivupyynnössä lähetetään palvelimen asettama keksi

Verkkoliikenteestä voidaan päätellä siirrettävien keksien arvot helposti, mutta saman tiedon saa myös suoraan selaimesta seuraavalla koodilla:

```
javascript:alert(document.cookie)
```

Koodi 15: Javascript-komento, jolla saa tietää sivulla käytössä olevat keksit

Kuva 23: Selaimen palauttama tieto sivulla käytössä olevista kekseistä

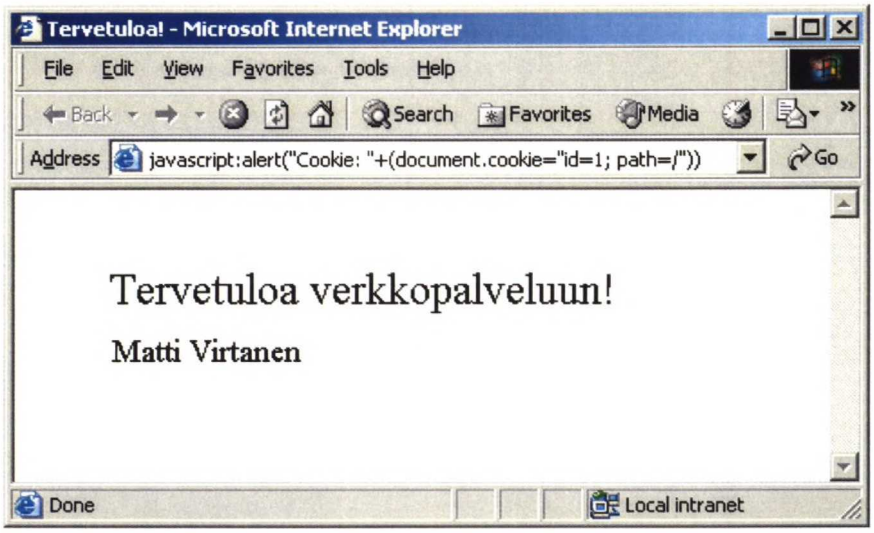
Kyseisen keksin nimestä ja arvosta voi päätellä, että sillä erotetaan käyttäjät toisistaan, joten erittäin todennäköisesti on mahdollista hyökätä keksin arvoa muuttamalla ja saada näin toisen henkilön identiteetti. Seuraavalla koodilla muutetaan selaimen keksien arvoja:

```
javascript:alert("Cookie: "+(document.cookie="id=1;
path=/"))
```

Koodi 16: Javascript-koodi, jolla muutetaan sivun käytössä olevaa keksiä

Kuva 24: Selaimen näkymä, kun vaihdettiin käytössä olevaa keksiä

Tästä eteenpäin selain palauttaa uuden keksin arvon palvelimelle jokaisella sivupyynnöllä aivan samalla tavoin kuin aikaisemminkin. Seurauksena on se, että palvelin luulee henkilöä toiseksi, kuten *kuvasta 25* näkyy.



Kuva 25: Palvelin erehtyy luulemaan henkilöä toiseksi keksin perusteella

Esimerkki havainnollisti keksien haavoittuvuuden ja niihin luottamisessa piilevään vaaran. Tällaista hyökkäystä voidaan käyttää esimerkiksi omien käyttöoikeuksien korottamiseen, ts. tekeydytään korkeammat käyttöoikeudet omaavaksi henkilöksi.

4.3.3.4 Parannuskeinoja

Lomakkeiden arvojen väärentämisen pystyy estämään tarkastamalla jokaisen asiakkaalta saadun arvon käytössä olevia vaihtoehtoja vastaan. Jos arvo on sallittu kyseiselle käyttäjälle kyseisessä tilanteessa, voi sitä käyttää sovelluksessa. Muussa tapauksessa pitää käyttää joko aikaisemmin valittuna ollutta arvoa, oletusarvoa, tai näyttää käyttäjälle virheilmoitus kielletyn arvon käyttämisestä.

Keksien väärentämistä vastaan kannattaa käyttää ns. istuntokohtaisia tunnuksia, jotka on generoitu jollakin sellaisella algoritmilla, ettei toisen ihmisen tunnusta voida arvata. Useissa ohjelmointikielissä tähän on tuki jo suoraan, mutta ellei ole, tulee sellainen algoritmi tehdä itse. Generoidun merkkijonon tulee olla tarpeeksi pitkä, ettei sen arvaaminen ole kohtuullisessa ajassa mahdollista. Tunnistuksessa käytettävän keksin eliniän tulisi olla myös mahdollisimman pieni, mielellään vain kyseisen istunnon pituinen, varsinkin jos sovelluksessa käytetään eri turvatasoja. Keksin eliniän ollessa pitkä riski siihen, että toisen keksin arvo kaapataan verkkoliikenteestä, kasvaa. Jos keksin kaappaamista ei havaita millään muulla suojauskeinolla, kuten esimerkiksi ip-osoitteen tarkastamisella, ei ole mitään merkitystä onko keksissä käytetty pitkää ja monimutkaista merkkijonoa vai ei, koska hyökkääjä voi käyttää saamaansa keksiä sen eliniän ajan. Verkkoliikenteen suojaamiseen voi käyttää salattua SSL/TLS-yhteyttä, mutta harvassa tilanteessa sen jatkuva käyttäminen on mielekästä. Salattua yhteyttä kannattaa käyttää ainakin sisäänkirjautumisen yhteydessä, jolloin henkilön tunnistamiseen tarvittava data siirretään salattuna.
([6], s. 435-437; [7], s. 19-20, 31, 36-37, 39-40)

4.3.4 Cross-site scripting ja selaimen ohjelmointi

4.3.4.1 Kuvaus

Cross-site scripting (lyh. XSS) on hyökkäys, jolla pystytään suorittamaan käyttäjän selaimessa ohjelmakoodia - joko javascript:iä tai vbscript:iä - hänen ollessaan yhteydessä luotettuun web-sivustoon. Koska ohjelmakoodi saadaan luotettavasta lähteestä, selain ei voi mitenkään erottaa hyökkäyskoodia muista koodeista. Hyökkäyskoodilla on pääsy mm. käyttäjän tunnistamisessa käytettyihin evästeisiin, jotka hyökkääjä pyrkii lähettämään itselleen. Samalla tavalla

voidaan myös muuttaa luotetun web-sivuston sisältöä ja näin antaa käyttäjälle väärä kuva sivuston sisällöstä.

([6], s. 413-417; [7], s. 26-27)

4.3.4.2 Esimerkki

Esimerkkisovellus vastaa minimaalista kuvitteellisen yrityksen kotisivua, jossa halutaan näyttää yritystä koskevia uutisia. Sivulla näytetään myös sisäänkirjautuneen käyttäjän nimi, joka on välitetty sivulle kirjautumisen jälkeen parametrissa. Esimerkki on *kuvassa 26*.

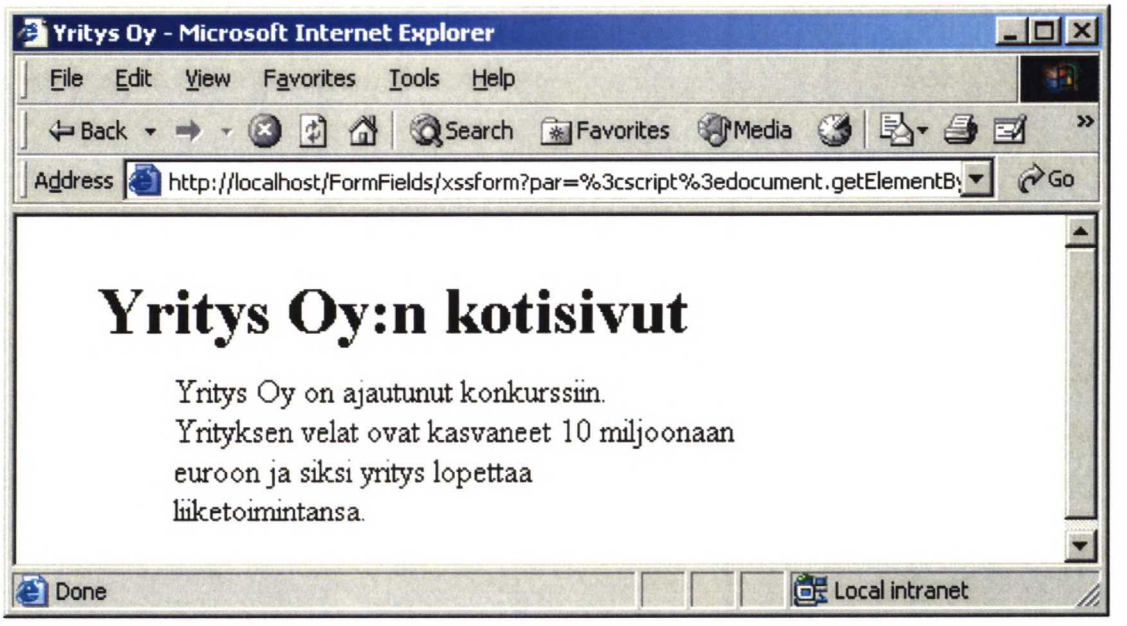
Kuva 26: Esimerkki yksinkertaisesta kotisivusta

Sivu on toteutettu siten, että parametrin arvo kirjoitetaan suoraan sivulle. Hyökkääjä voi käyttää sitä hyödykseen ja muuttaa sivulla olevaa uutista haluamakseen. Hyökkäys on *koodissa 17* ja toimii, kun se sijoitetaan sivun odottaman parametrin paikalle.

```
<script>document.getElementById(\"labNews\").innerText=\"Yritys Oy
on ajautunut konkurssiin. Yrityksen velat ovat kasvaneet 10 mil-
joonaan euroon ja siksi yritys lopettaa liiketoimintansa.\"</script>
```

Koodi 17: Hyökkäys, jolla vaihdetaan sivulla olevaa uutistekstiä

Hyökkäyksen seurauksena sivu muuttuu seuraavanlaiseksi:



Kuva 27: Hyökkäyksen muuttama websivu

Nyt hyökkääjä voi jakaa muodostettua linkkiä muille tai linkittää sivua muihin sivuihin, jolloin muut näkevät sen sisällön vääristyneenä. Hyökkääjä voi muuttaa parametria vaikeammin luettavaan muotoon url-koodaamisen (engl. url-encode) avulla. Tällöin on kokeneemmankin käyttäjän vaikea saada selville mitä parametrissa itse asiassa välitetään. *Koodissa 18* on esimerkki url-koodauksesta, joka on tehty *koodin 17* hyökkäykselle.
([6], s. 416-417)

```
%3c%73%63%72%69%70%74%3e%64%6f%63%75%6d%65%6e%74%2e%67%65%74%45%6c%6
5%6d%65%6e%74%42%79%49%64%28%22%6c%61%62%4e%65%77%73%22%29%2e%69%6e%
6e%65%72%54%65%78%74%3d%22%59%72%69%74%79%73%20%4f%79%20%6f%6e%20%61
%6a%61%75%74%75%6e%75%74%20%6b%6f%6e%6b%75%72%73%73%69%69%6e%2e%20%5
9%72%69%74%79%6b%73%65%6e%20%76%65%6c%61%74%20%6f%76%61%74%20%6b%61%
73%76%61%6e%65%65%74%20%31%30%20%6d%69%6c%6a%6f%6f%6e%61%61%6e%20%65
%75%72%6f%6f%6e%20%6a%61%20%73%69%6b%73%69%20%79%72%69%74%79%73%20%6
c%6f%70%65%74%74%61%61%20%6c%69%69%6b%65%74%6f%69%6d%69%6e%74%61%6e%
73%61%2e%22%3c%2f%73%63%72%69%70%74%3e
```

Koodi 18: Hyökkäys, jolla vaihdetaan sivulla olevaa uutistekstiä url-koodatussa muodossa

Samanlaisella menetelmällä voitaisiin tehdä myös hyökkäys, joka lähettää käyttäjän selaimen kaikki evästeet hyökkääjälle. Evästeiden avulla hyökkääjä voisi saada käyttäjän oikeudet omaan käyttöönsä.
([6], s. 416-417)

4.3.4.3 Parannuskeinoja

Parannuskeinoja mietittäessä saattaa käydä mielessä salauksen käyttäminen, mutta sillä ei pystytä lieventämään tätä uhkaa, koska ongelma johtuu huonosta syötteen käsittelystä. Koska hyökkäys voi syntyä parametrien välittämisestä, lomakkeiden arvoista, http:n otsikkotiedoissa, evästeissä tai tietokannan sisällöstä, on sovelluksen suunnittelussa oltava erittäin tarkkana, jotta hyökkäystietoa ei sellaisenaan lähetetä eteenpäin. Jokainen paikka, josta tietoa luetaan sisään, tulee käydä tarkasti läpi ja analysoida, missä kyseistä tietoa myöhemmin näytetään. Jos sitä näytetään jossain, tulee tiedon sisältö tarkistaa ennen kuin se talletetaan tai sitä käytetään uudestaan.
([6], s. 417,431)

4.3.5 SQL-injektiot

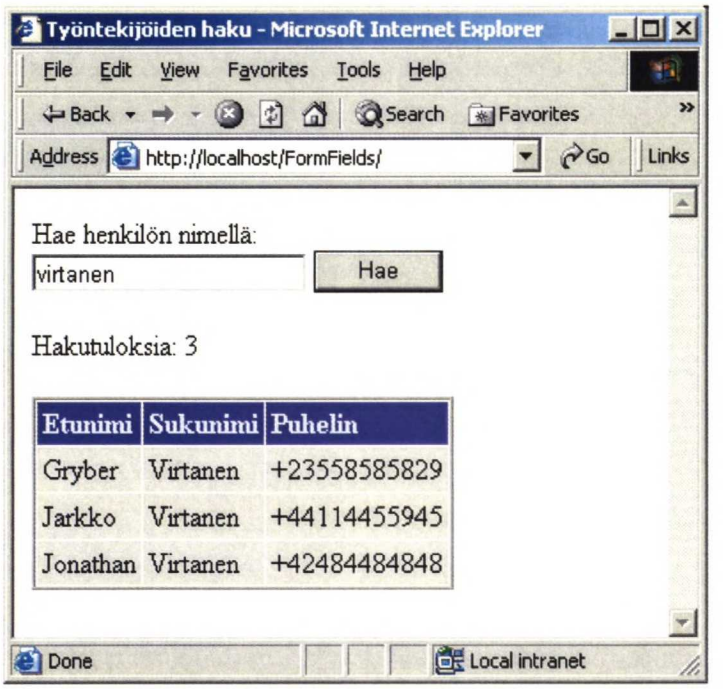
4.3.5.1 Kuvaus

SQL-injektiot ovat kasvaneet webin käytön lisääntymisen myötä suureksi uhaksi sovelluksille. SQL-injektiossa hyökkääjä onnistuu syöttämään (injektoimaan, engl. injection) sovelluksen käyttämiin SQL-lauseisiin omia haitallisia komentojaan. Niillä hän voi tehdä kaiken mitä sovelluksen ohjelmoijakin, kuten esim. käsitellä tietokantaa SQL:n avulla, kutsua tallennettuja komentoja (engl. stored procedure) tai kutsua jopa ulkoisia ohjelmia. Vakavaksi uhaksi SQL-injektion nostaa se, että harva ohjelmoija todella ymmärtää hänenkin tekemänsä sovelluksen olevan haavoittuvainen tälle hyökkäykselle. Moni toimiva SQL-kysely voi väärällä syötteellä toimia täysin päinvastoin kuin tarkoitettu. SQL-injektiot ovat ohjelmointikieli-, tietokanta- ja käyttöjärjestelmäriippumattomia, joten uhka koskee monia hyvin erilaisilla tekniikoilla toteutettuja sovelluksia. Uhka koskee websovellusten lisäksi myös muita sovelluksia, joissa käytetään tietokantoja, mutta niiden uhka on pienempi,

koska käyttäjäkunta on rajattu esim. yritysten omiin työntekijöihin.
([6], s. 398-399, [20])

4.3.5.2 Esimerkki

Esimerkkisovellus SQL-injektiosta on yksinkertainen työntekijöiden haku. Sovelluksessa on syöttökenttä, johon käyttäjä voi kirjoittaa hakemansa henkilön nimen, tai vain osan siitä. Syötetyllä hakuehdolla haetaan työntekijöitä SQL-tietokannasta, ja haun tulokset palautetaan käyttäjälle.



Kuva 28: Työntekijän haku

Kuvassa 28 näkyy tilanne käyttäjän haettua työntekijöitä hakusanalla "virtanen". Tietokannasta on löytynyt kolme henkilöä, joiden tiedot näytetään taulukossa. Selain lähettää seuraavan pyynnön palvelimelle:

```
POST /FormFields/db HTTP/1.1
...
Content-Length: 36

texSearchText=virtanen&btnSearch=Hae
```

Koodi 19: Selaimen palvelimelle lähettämä HTTP-kutsu

Palvelinohjelma vastaanottaa selaimen lähettämän kutsun ja hakee käyttäjän syöttämän hakusanan "texSearchText"-muuttujasta. Muuttujan arvo kulkeutuu sovelluksen hakutoimintoon seuraavasti:

```
1    select Firstname, Lastname, Phonenumber
2    from users
3    where Lastname like '" + texSearchText + "%' or
4          Firstname like '" + texSearchText + "%'
```

Koodi 20: Haussa käytetty SQL-komento

Kun *koodin 20* mukaiseen SQL-komentoon lisätään käyttäjän syöttämä hakusana, saadaan lopulta *koodin 21* mukainen SQL-haku:

```
1    select Firstname, Lastname, Phonenumber
2    from users
3    where Lastname like 'virtanen%' or
4          Firstname like 'virtanen%'
```

Koodi 21: Hakukomento käyttäjän syötteellä

Kyseinen komento palauttaa tietokannasta kolme käyttäjää, kuten *kuvassa 28* näkyi. Hyökkääjä voi siis injektoida haluamiaan komentoja suoraan tietokantamoottorille suoritettavaksi. *Koodissa 22* näkyy eräs mahdollinen syöte, jota voidaan käyttää hyökkäyksenä.

```
1    '; drop table users --
```

Koodi 22: Hyökkääjän syöttämä arvo hakukenttään ([7], s. 28)

Hyökkääjän syöte yhdistettynä sovelluksen hakukomentoon tuottaa seuraavan komennon:

```
1    select Firstname, Lastname, Phonenumber
2    from users
3    where Lastname like ''; drop table users -- or
8          Firstname like ''; drop table users --
```

Koodi 23: Hakukomento vihamielisellä syötteellä

Kyseinen koodi poistaa tietokannasta kyseisen taulun kokonaan, joten käyttäjien tiedot eivät enää ole käytettävissä. Hyökkäyksessä käytettiin apuna "--" – kommenttimerkkijonoa, jolloin hakukomennon loppuosa jäi kokonaan huomioimatta. Kommenttimerkin käyttämisen voi myös kiertää, kuten *koodissa 24* näkyy, jolloin pyritään muodostamaan koko kyselystä täydellinen SQL-lause. Koodissa kuvattu hyökkäys poistaa kaikki käyttäjät tietokannasta.

```
1   '; delete from users; select * from users where lastname = '
```

Koodi 24: Hyökkääjän syöttämä arvo hakukenttään, jolla muodostetaan täydellinen SQL-lause

4.3.5.3 Parannuskeinoja

SQL-injektiota varten voi suojautua tarkistamalla kaikki syötteet ennen kuin ne lähetetään tietokannalle. Tämä on paras puolustus SQL-injektioita vastaan. Lisäksi kannattaa käyttää parametrisoituja tallennettuja komentoja, joita vastaan on vaikeampi hyökätä. Myös hakua varten kannattaa luoda käyttäjätunnus, jolla on minimaaliset käyttöoikeudet tietokantaan ja sen tauluihin. Jos sovellus tarvitsee vain muutamaan tauluun lukuoikeudet, tulee niille tauluille antaa vain lukuoikeus ja evätä kokonaan pääsy muihin tauluihin. Näin pystytään hyökkäystilanteessa minimoimaan tapahtunut vahinko, kun hyökkääjä ei voi tehdä lukuja, päivityksiä tai poistoja muihin tauluihin. Ohjelmointikieliin on myös lisätty parannuksia, joilla pystytään heikentämään tai poistamaan kokonaan SQL-injektiouhka, joten valitun ohjelmointikielen tarjoamat mahdollisuudet tulee tarkistaa tietokantaa käyttävää sovellusta tehdessä.
([7], s. 28)

4.3.6 Palvelunestohyökkäys

4.3.6.1 Kuvaus

Palveluestohyökkäyksessä (engl. denial of service, dos) hyökkääjä pystyy toimillaan saamaan palvelun siihen tilaan, ettei se ole muiden käytettävissä. Palvelunestohyökkäys voidaan toteuttaa myös hajautetusti (engl. distributed dos, ddos), jolloin useampi kone hyökkää palveluun samanaikaisesti. Palvelunestohyökkäyksellä olla monta eri tavoitetta, kuten verkkokaistan, muistin, proses-

sorin tai resurssien kuormittaminen. Hyökkäyksellä voidaan saada aikaan myös sovelluksen tai käyttöjärjestelmän kaatuminen. Palvelunestohyökkäystä vastaan on kaikkein hankalin puolustautua. Siinä onnistumiseksi on määriteltävä, kuinka sovellusta vastaan voidaan hyökätä ja miten se voidaan estää. Monet vähättelevät palvelunestohyökkäyksen uhkia, mutta tosiasia on, että sitä voidaan käyttää palvelimen toiminnan kaappaamiseen jos alkuperäinen palvelin ei ole käytettävissä.
([1], s. 510-514; [6], s. 517; [11], s. 304-306)

4.3.6.2 Esimerkki prosessorin kuormittamisessa

Esimerkkiohjelmassa, joka on kokonaisuudessaan liitteessä 2, osoitetaan kuinka suuri ero suorituskyvyssä voi olla eri koodien välillä syötteen kasvaessa suureksi. Ohjelmassa on toteutettu "\\"-merkkien korvaaminen yhdellä "\"-merkillä. Kyseistä algoritmia voidaan käyttää esimerkiksi tiedostonimien korjaamisessa tilanteessa, jossa käyttäjä on syöttänyt virheellisesti ylimääräisiä kenoviivoja. *Koodissa 25* näkyy ensimmäinen toteutus algoritmista.
([6], s. 521-529)

```
1   bool StripBackslash1(char* buf)
2   {
3       char* tmp = buf;
4       bool ret = false;
5       for(tmp = buf; *tmp != '\0'; tmp++)
6       {
7           if(tmp[0] == '\\' && tmp[1] == '\\')
8           {
9               strcpy(tmp, tmp+1);
10              ret = true;
11          }
12      }
13      return ret;
14  }
```

Koodi 25: Kaksinkertaisten kenoviivojen poistoalgoritmin ensimmäinen toteutus
([6], s. 522)

Koodin toiminta on toteutettu siten, että puskuria käydään läpi tavu kerrallaan ja jos sekä läpikäytävässä sekä sitä seuraavassa kohdassa on kenoviiva, kopioidaan merkkijono itsensä päälle niin, että toinen kenoviiva poistuu. Toinen toteutus samasta algoritmista on *koodissa 26*.

```
1   bool StripBackslash2(char* buf)
2   {
3       unsigned long len = strlen(buf)+1, written=0;
4       char* tmpbuf = (char*)malloc(len);
5       char* tmp;
6       bool foundone = false;
8       for(tmp = buf; *tmp != '\0'; tmp++)
9       {
10          if(tmp[0] == '\\' && tmp[1] == '\\')
11              foundone = true;
12          else
13          {
14              tmpbuf[written] = *tmp;
15              written++;
16          }
17      }
18      if(foundone)
19      {
20          strncpy(buf, tmpbuf, written);
21          buf[written] = '\0';
22      }
23      if(tmpbuf != NULL)
24          free(tmpbuf);
25      return foundone;
26  }
```

Koodi 26: Kaksinkertaisten kenoviivojen poistoalgoritmin toinen toteutus (lyhennetty alkuperäisestä) ([6], s. 523-524)

Toisessa toteutuksessa toiminto perustuu apupuskurin käyttöön. Alkuperäistä merkkijonoa käydään läpi merkki merkiltä, ja jos peräkkäiset merkit eivät ole kenoviivoja, merkki kopioidaan apupuskuriin. Kun merkkijono on tutkittu kokonaisuudessaan, apupuskuri kopioidaan alkuperäisen puskurin päälle. Tällä toteutuksella pystytään välttämään jatkuvasti tapahtuva merkkijonojen kopiointia. Kun merkkijonon pituutta kasvatetaan, voidaan huomata näiden mene-

telmien välillä oleva suoritusero, kuten *taulukossa 3* esitetään. Taulukossa on mukana kolmaskin "StripBackslash3"-funktio, listattu liitteessä 2, joka on suorituskyvyltään paras esitetyistä vaihtoehdoista. Taulukossa tähdillä merkittyä testiä ei ole suoritettu, koska se on kohtuuttoman pitkäkestoinen suoritettavaksi.

Pituus merkkeinä	Toteutus 1 (ms)	Toteutus 2 (ms)	Toteutus 3 (ms)
10	0	0	0
100	0	0	0
1 000	0	0	0
10 000	70	0	0
100 000	6 119	0	0
1 000 000	835 581	20	10
10 000 000	**********	190	80

Taulukko 3: Kaksinkertaisten kenoviivojen poistoalgoritmien nopeusvertailu ([6], s.525)

Jos hyökkääjä siis onnistuu syöttämään pitkän merkkijonon hitaampaan toteutukseen, pystyy hän lamaannuttamaan palvelimen hyvin pitkäksi aikaa. Tänä aikana palvelin ei pysty palvelemaan muita käyttäjiä tehokkaasti, tai pahimmassa tapauksessa ollenkaan.

4.3.6.3 Parannuskeinoja

Sovelluksen kaatumiseen johtavat virheet viittaavat aina huonoon koodin laatuun. Sitä voidaan parantaa koodin läpikäynneillä ja testaamisella. Prosessorin ylikuormitushyökkäykset ovat suorituskykyyn liittyviä ongelmia, jotka voidaan havaita profiloimalla koodin suoritusta silloin, kun sovellukselle on annettu hyökkääviä syötteitä. Profiloinnista voidaan päätellä, missä kohdissa ohjelmaa kuluu eniten aikaa, minkä jälkeen voidaan parantaa näiden kohtien suorituskykyä. Muistin ja resurssien ylikuormitusongelmat syntyvät sovelluksen suunnitteluvaiheessa. Parannuskeinoina näihin toimivat suojausmekanismit, jotka havaitsevat hyökkäykset ja reagoivat niihin muuttamalla sovelluksen toimintaa.

Kaistan kuormittamista vastaan tehtyihin hyökkäyksiin voi löytää parannuskeinoja tutkimalla, miten sovellus vastaa verkosta tuleviin sopimattomiin kutsuihin.
([6], s. 533)

4.3.7 Verkkokuuntelu ja väärennökset

4.3.7.1 Kuvaus

Sovellusten on tunnistettava käyttäjänsä, jotta heille voidaan tarjota personoituja palveluita tai palveluita, jotka eivät ole kaikille sallittuja. Siksi käyttäjä tunnistetaan yleensä käyttäjätunnuksen ja salasanan avulla. Kun käyttäjä lähettää verkon kautta sovellukselle käyttäjätunnuksensa ja salasanansa, voi jokin kolmas osapuoli saada tiedot käsiinsä salakuuntelemalla verkkoliikennettä. Tällöin ulkopuolinen voi käyttää käyttäjältä saatuja tunnuksia kuten omiaan. Verkkokuuntelulla voidaan päästä käsiksi muihinkin arkaluontoisiin tietoihin, joita voidaan käyttää vihamielisiin tarkoituksiin.
([12], s. 22-23)

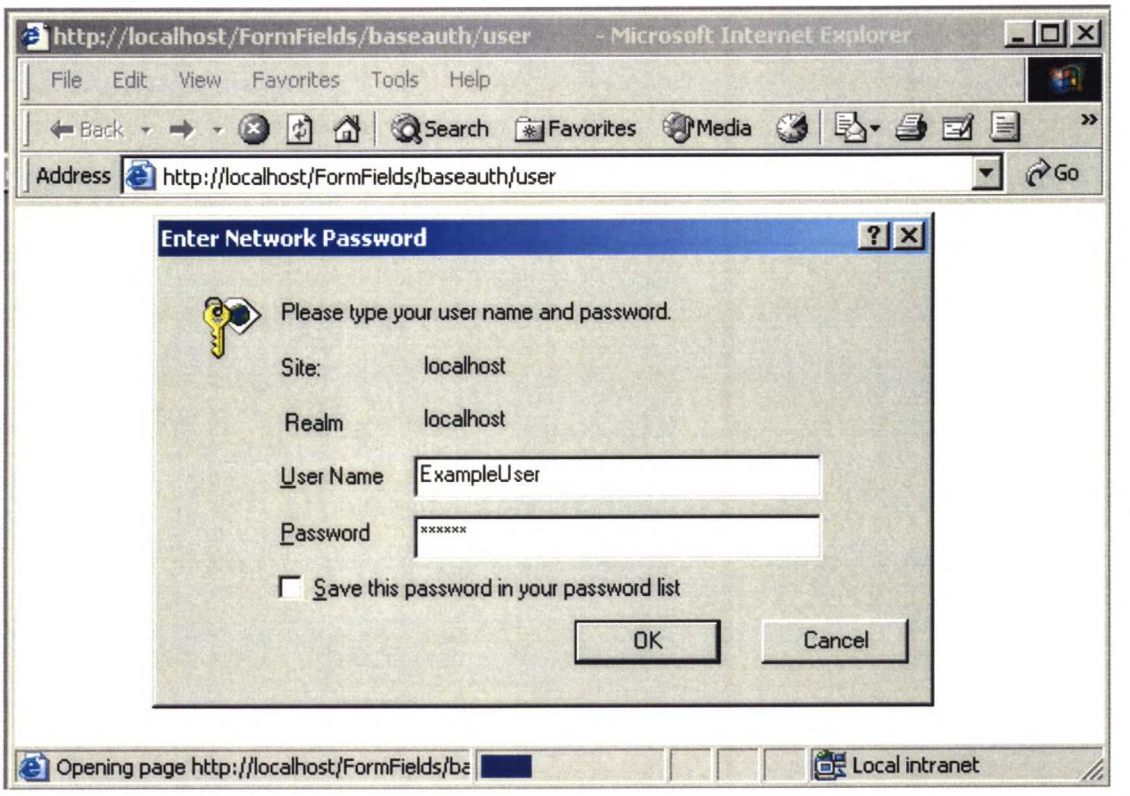
4.3.7.2 Esimerkki http:n tunnistautumisesta

Esimerkkisovelluksessa osoitetaan http:n peruskäyttäjäntunnistuksen (engl. basic authentication) vaarallisuus. Kun käyttäjä hakee suojattua sivua, palvelin palauttaa käyttäjälle *koodin 27* mukaisen sivun. Rivillä 4 näkyy tunnistautumispyyntö, jossa mainitaan käytettäväksi menetelmäksi perustyypin tunnistautuminen.

```
1  HTTP/1.1 401 Access Denied
2  Server: MyServer
3  Date: Tue, 10 Aug 2004 04:59:17 GMT
4  WWW-Authenticate: Basic realm="localhost"
5  Connection: close
6  Content-Length: 4431
7  Content-Type: text/html
8
9  <html>...
```

Koodi 27: Palvelimen tunnistautumispyyntö käyttäjälle

Käyttäjälle näytetään tällöin sisäänkirjautumisikkuna, johon käyttäjä voi syöttää omat tunnuksensa, kuten *kuvassa 29* näkyy.



Kuva 29: Käyttäjälle näytettävä sisäänkirjautumisikkuna

Kun käyttäjä on kirjoittanut tunnuksensa, lähettää selain seuraavan pyynnön palvelimelle:

```
1   GET /FormFields/BaseAuth/user HTTP/1.1
2   User-Agent: Mozilla
3   Host: localhost
4   Connection: Keep-Alive
5   Authorization: Basic RXhhbXBsZVVzZXI6JGVjcjN0
```

Koodi 28: Käyttäjän selaimen lähettämä sivupyyntö palvelimelle tunnistautumistietoineen

Rivillä 5 näkyy selaimen lähettämä base64-menetelmällä koodattu vastaus. Palvelin käsittelee kyseisen tiedon ja varmistaa käyttäjän tunnuksen. Jos tunnus on oikein, palvelin palauttaa käyttäjän pyytämän sivun. Ongelmana on, et-

tä perusmenetelmässä käyttäjätunnus ja salasana lähetetään base64-menetelmällä koodattuna, ei salattuna. Base64-menetelmä voidaan kääntää takaisin selväkieliseksi tekstiksi pelkästään tietämällä base64:lla käsitelty merkkijono. Kun *koodin 28* mukaisen base64-merkkijonon muuttaa takaisin selväkieliseksi tekstiksi, tuloksena on seuraava merkkijono:

```
ExampleUser:$ecr3t
```

Koodi 29: Base64-merkkijono muutettuna selväkieliseksi

Selväkielisestä merkkijonosta voi päätellä suoraan käyttäjätunnuksen ja salasanan. Jos hyökkääjä saa kaapattua käytetyn liikenteen koneiden välillä, voi hän käyttää saamiaan vieraita tunnuksia vapaasti omiin tarkoitusperiinsä.

4.3.7.3 Parannuskeinoja

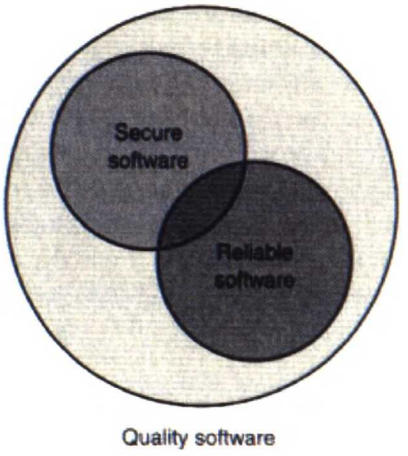
Käyttäjätunnuksen ja salasanan siirtämisessä verkon yli tulee aina käyttää salattua yhteyttä tai vähintään siirrettävään tiedon salausta, vaikka yhteys ei olisikaan salattu. Perustunnistautumista ei tulisi koskaan käyttää salaamattomassa yhteydessä, koska hyökkääjä pystyy salakuuntelemalla saamaan käyttäjän tunnuksen ja salasanan haltuunsa, aivan kuten edeltävässä esimerkissäkin näytettiin. Verkkokuunteluun tulee muutenkin varautua, etteivät salakuuntelijat saa tärkeitä tietoja kaapattua salaamattomasta yhteydestä.
([6], s. 437; [12], s. 22-23)

4.4 Yhteenveto

Tässä luvussa esiteltiin yleisiä tietoturvaongelmia ja hyökkäyksiä niitä vastaan. Tämän luvun perusteella pitäisi tulla selväksi se, kuinka vaarallisia kyseiset ohjelmointivirheet tai suunnitteluvirheet voivat olla. Tietoturvaongelmia aiheuttavia virheitä on paljon enemmän kuin tässä käsiteltiin, mutta luvussa pyrittiinkin antamaan vain yleiskuva siitä, millaisia virheet ja hyökkäykset voivat olla.

5 Yhteenveto

Ohjelmointivirheet, jotka aiheuttavat yleisimmät tietoturvaongelmat, ovat lähes poikkeuksetta syötteen tarkistukseen liittyviä. Syötettä ei joko tarkisteta ollenkaan tai se tarkistetaan huolimattomasti. Seuraukset voivat olla katastrofaaliset, kuten edellisessä luvussa nähtiin. Ohjelmoijat ajattelevat, että "tästä tiedän vain minä", mikä johtaa siihen, ettei kyseiseen ongelmaan panosteta riittävästi. Tietoturvavälikohtausten määrälle ei kuitenkaan ole näkyvissä vähenemisen merkkejä. Myöskään markkinoille ei ole tullut uutta keksintöä, jolla pystyttäisiin vähentämään merkittävästi nykyisten ohjelmointivirheiden aiheuttamia ongelmia. Kun nämä tosiasiat huomioidaan, sovelluskehityksen tason nostamiseen ei varmasti ole mitään muuta keinoa kuin parantaa ohjelmoijien tietämystä tietoturvasta ja tietoturvaongelmien syistä. Sovelluskehittäjien on tärkeää oivaltaa, että heidän sovelluksiaan tullaan tarkoituksella käyttämään väärin, joten heti suunnitteluvaiheessa tulee huomioida asiaankuuluvat lisävaatimukset. Tavoitteena on laadukkaat sovellukset, joiden luotettavuus ja tietoturva ovat korkealla tasolla, kuten *kuvassa 30* näkyy:



Kuva 30: Sovellusten tavoite: turvallisuus ja luotettavuus ([6], s. 9)

Tietoturvatavoitteeseen päästään, kun jokainen tekijä tiedostaa tietoturva-aukkojen takana olevan perimmäisen syyn (esitetty tarkemmin luvussa 2.3.3):

Kaikki syöte on pahaa syötettä ([6], s. 341)

Jokaisen sovelluksen tulee pitää syötettä vihamielisenä hyökkäyksenä, kunnes sen todistetaan olevan sallittua ja vaaratonta. Jos sovellukset hallitsisivat syötteen tarkistukset kunnolla, emme kuulisi matojen ja virusten leviämisestä niin usein kuin tällä hetkellä kuulemme.

6 Pohdinta ja johtopäätökset

Diplomityössä käsiteltiin tietoturvaa sovellusohjelmoijan näkökulmasta. Tietoturvaongelmat havainnollistettiin esimerkkien avulla. Jokainen ongelma käytiin läpi niin tarkasti, että sen taustat tulevat sovellusohjelmoijalle selviksi, kuten myös kyseistä haavoittuvuutta hyväkseen käyttävän hyökkäyksen seuraukset. Esimerkeissä olisi voitu selventää myös, kuinka kyseinen virhe vältetään tai kuinka ohjelmaa tulisi muuttaa, jotta ongelma poistuu. Lisäksi olisi voitu esitellä uusia tekniikoita, joiden avulla kyseinen ongelma poistuu kokonaan tai sen uhka vähenee merkittävästi. Se olisi kuitenkin kasvattanut esimerkkien kokoa tarpeettomasti, eikä hyöty olisi ollut kovin suuri. Suurin osa ohjelmoijista kuitenkin ymmärtää, miten sovellusta tulee muuttaa virheen poistamiseksi, kunhan heille ensin selvitetään kyseisen ongelman taustat esimerkin avulla.

Työn alkuvaiheessa tietoturvaa käsiteltiin yleisellä tasolla, minkä tarkoitus oli toimia johdantona työn tärkeimmälle osalle, tietoturvaongelmia esitteleville esimerkeille. Tämä osuus on koottu useasta eri lähteestä ja useaa eri näkökulmaa ajatellen. Seurauksena on hieman hajanainen kokonaisuus, joka nostaa esiin yksittäisiä asioita ilman yhteen kokoavaa punaista lankaa. Osuuden täydellinen uudistaminen ei kuitenkaan palvele tarkoitusperää - esitellä tietoturvaongelmia sovellusohjelmoijille - joten osuus jää irrallisten, joskin tärkeiden asioiden koontiosaksi.

Tietoturvaongelmien esittäminen ei pelkästään riitä. On pystyttävä kehittämään keinot niiden estämiseksi jatkossa kokonaan. Se vaatii ohjelmoijien kouluttamista tietoturvallisten sovellusten tekemiseen ja suunnittelemiseen. Kouluihin tulisi lisätä ohjelmointikursseja, joissa perehdytään tietoturvaan tarkemmin. Kurssien tulisi olla enemmän käytännöllisiä kuin teoreettisia, mikä antaisi paremmat keinot ratkaista haastavia ohjelmointiongelmia kunnollisen tietoturvan saavuttamiseksi. Sovellusten suunnittelussakin on panostettu enemmän hienojen oliomallien toteuttamiseen sen sijaan, että luotaisiin ylläpidettäviä ja pitkäikäisiä hyviä sovelluksia.

7 Sovelluskehityksen tulevaisuus

Monet tietoturvaongelmiin johtaneet ohjelmointivirheet ovat erityisesti vanhojen ohjelmointikielien vitsauksia, kuten puskurin ylivuodot. Näitä virheitä on korjattu uusimpiin ohjelmointikieliin, jotka tarkkailevat puskurin käyttämistä ajonaikaisesti ja ilmoittavat, jos puskuriin yritetään sijoittaa liikaa dataa. Näin saadaan puskurin ylivuotoon liittyvät ongelmat siirrettyä tavalliselta ohjelmoijalta ohjelmointialustan tekijälle. Myös käyttöjärjestelmiin tehtävät suojaukset puskurin ylivuotoa vastaan parantavat sovelluksien tietoturvaa. Näillä keinoilla haavoittuvuuksien määrä putoaa jo kolmanneksella, koska puskurin ylivuodot ovat hyvin yleisiä ([21], s. 14).
([22], s. 79)

Web-ohjelmointiin on lisäksi kehitetty avuksi ns. komponenttimalli, jolla pystytään parantamaan tietoturvaa merkittävästi (esim. Microsoftin kehittämä .NET-arkkitehtuuri). Tässä mallissa ohjelmoijat eivät käytä enää web-ohjelmoinnin peruskomponentteja, kuten esimerkiksi valintalistoja, vaan uusia komponentteja, jotka toimivat kuten perinteisissä graafisissa ohjelmointikielissä. Näiden komponenttien avulla ohjelmoija käsittelee käyttäjän syötteitä aivan kuin ne olisivat graafisessa käyttöliittymässä tehtyjä. Etuna tästä on, ettei ohjelmoijan tarvitse enää hakea arvoja lomakkeen kentistä tai url-parametreista, joita hyökkääjän on helppo muokata. Kun komponentit sallivat arvoikseen vain niille sallittuja arvoja, ohjelmoija voi käyttää suoraan komponentin arvoja ilman pelkoa siitä, että kyseinen arvo on sallitun arvovälin ulkopuolella. Tämä mahdollistaa esimerkiksi valintalistojen turvallisemman käytön, kun valinnan arvoihin voi luottaa. Tässä mallissa vastuu tietoturvasta on siirretty hyvin pitkälle komponentin tekijälle, jolloin sovellusohjelmoijan ei tarvitse hallita tietoturvaohjelmointiin liittyviä erikoishaasteita. Vastaavalla tavalla myös web-sovelluksessa käytettävät yleiset toiminnot, kuten sisäänkirjautuminen, voidaan rakentaa valmiiksi komponenteiksi, jotka huomioivat jo valmiiksi kaikki kyseiseen toimintoon liittyvät tietoturvahaasteet. Saman mallin avulla voidaan toteuttaa myös tietokantaa käsittelevät toiminnot, joissa komponentti estää SQL-injektion tyyliset hyökkäykset. Tällöin sovellusohjelmoijan ei tarvitse

tuntea kaikkia mahdollisia keinoja hyökkäyksen estämiseksi, vaan hän voi käyttää apuna komponenttia, joka hallitsee asian jo valmiiksi. Komponentteihin voidaan rakentaa myös erilaisia lisäominaisuuksia, joilla pystytään parantamaan sovelluksen tietoturvaa muiltakin osa-alueilta. Tällaisia ovat esimerkiksi palvelunestohyökkäyksiä vastaan auttavat erilaiset välimuistit (engl. cache), joihin voidaan tallettaa usein käytettyä tietoa tai tietoa, jonka hakeminen kestää liian kauan sen hakemiseksi joka kerta, kun se tulisi näyttää. ([6], s. 404-405; [23]; [24])

Eräs keino parantaa tietoturvaa on ajaa sovelluksen osia mahdollisimman suppeilla oikeuksilla. Sovelluksen sisällä voidaan myös kieltää esimerkiksi tiedostojen luku ja kirjoitus, jolloin kyseinen moduuli ei voi suorittaa mitään tiedosto-operaatioita ajoympäristössä. Sovellukselle voidaan sallia myös tiedostojen käsittely vain omassa hakemistossa, mutta sen ulkopuolella sovelluksella ei ole oikeuksia. Tämä perustuu hiekkalaatikko (engl. sandbox) -ajatteluun, jossa sovellusta ajetaan eristetyssä ympäristössä, jonka sisäpuolella sovellus saa toimia vapaasti, mutta jonka ulkopuolelle sillä ei ole mitään asiaa. Hiekkalaatikkoa voidaan käyttää esimerkiksi selaimessa ajettaviin sovelluksiin, jotka saavat toimia vain hyvin rajoitetuilla oikeuksilla.
([6], s. 535-545; [23]; [24])

Lähteet

Tähdellä merkityt lähteet ovat saatavilla cd-levyllä.

[1] Scambray, Joel, McClure, Stuart, Kurtz, George. Osborne/McGraw-Hill. 2001. Hakkeroinnin torjunta: verkkoturvallisuuden salaisuudet ja ratkaisut

[2] Koziol, Jack, Litchfield, David, Aitel, Dave, Anley, Chris, Eren, Sinan, Mehta, Neel, Hassell, Riley. Wiley. 2004. The Shellcoder's Handbook: Discovering and Exploiting Security Holes.

[3] Erickson, Jon. No starch press. 2003. Hacking – The art of exploitation.

[4] * CERT Coordination Center. 11.5.2004. Incident Reporting System. https://irf.cc.cert.org/

[5] Eriksson, Hans-Erik, Penker, Magnus. IT Press. 2000. UML.

[6] Howard, Michael, LeBlanc, David. Microsoft Press. 2003. Writing secure code (2nd. Edition).

[7] * Meier, J.D., Mackman, Alex, Vasireddy, Srinath, Dunner, Michael, Escamilla, Ray, Murukan, Anandha. Microsoft Patterns & Practices. 2003. Improving Web Application Security: Threats and Countermeasures.

[8] Vaalisto, Heidi. ITviikko. 25.3.2004. Kauppiaat sulattelevat läskejään.

[9] * Suomen Pankkiyhdistys. 2004. Tilastotietoja pankkien maksujärjestelmistä Suomessa 1994-2003. http://www.pankkiyhdistys.fi/sisalto/upload/pdf/tilastot.pdf

[10] McGraw, Gary. Cigital. 6.1.2003. The Art and Science of Software Security. http://www.cigital.com/ssw/presentations/gem/index_files/frame.htm (11.5.2004).

[11] Järvinen, Petteri. Sanoma WSOY. 2002. Tietoturva & yksityisyys.

[12] Gollmann, Dieter. Wiley. 1999. Computer security.

[13] Paavilainen, Juhani. Suomen ATK-kustannus. 1998. Tietoturva.

[14] * CERT Coordination Center. 11.5.2004. Statistics 1988-2003. http://www.cert.org/stats/cert_stats.html

[15] * Microsoft Corporation. Microsoft Patterns & Practices. 2004. 21.6.2004. Threat Modeling. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod76.asp

[16] * One, Aleph. Phrack. 1996. 21.5.2004. Smashing The Stack For Fun And Profit. http://www.phrack.org/phrack/49/P49-14

[17] Kaspersky, Kris. A-list. 2003. Hacker disassembling uncovered.

[18] * Conover, Matt. w00w00 Security Development. 1999. 23.5.2004. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt

[19] * Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T. 1999. 12.7.2004. Hypertext Transfer Protocol -- HTTP/1.1. ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt

[20] SPI Dynamics, Inc. 2002. 21.5.2004. SQL Injection – Are Your Web Applications Vulnerable? http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf

[21] Saarelainen, Aki. Tietokone 10/2004. Windows XP sai viimein huolto-päivityksen.

[22] Haukilehto, Ahti. IT Press. 2003. Visual C# .NET.

[23] Thuan, Thai, Hoang, Lam. O'Reilly. 2001. .NET Framework essentials.

[24] Gunnerson, Eric. Springer Verlag Pub. 2000. A programmer's introduction to C#.

Liitteet

Liite 1: Keon ylivuotoesimerkki

Liite 2: Prosessorin kuormitusesimerkki ([6], s. 521-529)

Liite 1: Keon ylivuotoesimerkki

```
/*
Idea copied from example by
Jon Erickson in book:
Hacking: The art of exploitation (page: 41)
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[])
{
        char *login;
        char *username;

        if ( argc != 2 )
        {
                printf("Supply username as parameter\n");
                exit(1);
        }

        login = (char*)malloc(10);
        username = (char*)malloc(10);

        // Set current username for anonymous:
        strcpy( username, "anonymous" );

        // Get users loginname:
        strcpy( login, argv[1] );

        ////////////////////////////////
        // Here are login checks
        // ...
        // Login name can be copied to
        // username of changed to
        // match correct one
        ////////////////////////////////

        // Check users rights from current username:
        // (Of course you could see "admin" from binary
        // code.. but this is only example)
        if ( strcmp( username, "admin" ) == 0 )
        {
                // User is admin!
                printf("Welcome system administrator!\n");
                ////////////////////
                // Do admin tasks...
                ////////////////////
        }
        else
        {
                printf("Access denied!\n");
        }

        fflush(stdout);

        free( login );
        free( username );

        return 0;
}
```

Liite 2: Prosessorin kuormitusesimerkki ([6], s. 521-529)

```
/*
  Howard, Michael, LeBlanc, David:
  Writing secure code (2nd edition)

  ISBN: 0-7356-1722-8
*/
/*
  CPU_DoS_Example.cpp
  This application shows the effects of two
  different methods of removing duplicate backslash
  characters.

  There are many, many ways to accomplish this task. These
  are meant as examples only.
  */

#include <windows.h>
#include <stdio.h>
#include <assert.h>

/*
  This method reuses the same buffer, but is inefficient.
  The work done will vary with the square of the size of the input.

  It returns true if it removed a backslash.
  */

//We're going to assume that buf is null-terminated.
bool StripBackslash1(char* buf)
{
    char* tmp = buf;
    bool ret = false;

    for(tmp = buf; *tmp != '\0'; tmp++)
    {
        if(tmp[0] == '\\' && tmp[1] == '\\')
        {
            //Move all the characters down one
            //using a strcpy where source and destination
            //overlap is BAD! This is an example of how NOT to do things.
            //This is a professional stunt application - don't try this
            //at home.
            strcpy(tmp, tmp+1);
            ret = true;
        }
   }

    return ret;
}

/*
  This is a less CPU-intensive way of doing the same thing.
  It will have slightly higher overhead for shorter strings due to
  the memory allocation, but we only have to go through the string once.
  */

bool StripBackslash2(char* buf)
{
    unsigned long len, written;
    char* tmpbuf = NULL;
    char* tmp;
    bool foundone = false;

    len = strlen(buf) + 1;

    if(len == 1)
        return false;

    tmpbuf = (char*)malloc(len);

    //This is less than ideal - we should really return an error.
    if(tmpbuf == NULL)
    {
        assert(false);
        return false;
   }

    written = 0;
```

```
    for(tmp = buf; *tmp != '\0'; tmp++)
    {
        if(tmp[0] == '\\' && tmp[1] == '\\')
        {
            //Just don't copy this one into the other buffer.
            foundone = true;
        }
        else
        {
            tmpbuf[written] = *tmp;
            written++;
        }
    }

    if(foundone)
    {
        //Copying the temporary buffer over the input
        //using strncpy allows us to work with a buffer
        //that isn't null-terminated.
        //tmp was incremented one last time as it fell out of the loop.
        strncpy(buf, tmpbuf, written);
        buf[written] = '\0';
    }

    if(tmpbuf != NULL)
        free(tmpbuf);

    return foundone;
}


bool StripBackslash3(char* str)
{
char* read;
char* write;

//Always check assumptions.
assert(str != NULL);

if(strlen(str) < 2)
{
        //No possible duplicates.
        return false;
}

//Initialize both pointers.
for(read = write = str + 1; *read != '\0'; read++)
{
        // If this character and last character are both backslashes,
        // don't write - only read gets incremented.

        if(*read == '\\' && *(read - 1) == '\\')
        {
                continue;
        }
        else
        {
                *write = *read;
                write++;
        }
}

//Write trailing null.
*write = '\0';

return true;

}


int main(int argc, char* argv[])
{
    char* input;
    char* end = "foo";
    DWORD tickcount;
    int i, j;

    //Now we have to build the string.

    for(i = 10; i < 10000001; i *= 10)
```

```
    {
        input = (char*)malloc(i);

        if(input == NULL)
        {
            assert(false);
            break;
        }

        //Now populate the string.
        //Account for the trailing "foo" on the end.
        //We're going to write 2 bytes past input[j],
        //then append "foo\0".
        for(j = 0; j < i - 5; j += 3)
        {
            input[j] = '\\';
            input[j+1] = '\\';
            input[j+2] = 'Z';
        }

        //Remember that j was incremented before the conditional
        //was checked.
        strncpy(input + j, end, 4);

        tickcount = GetTickCount();
        StripBackslash1(input);
        printf("StripBackslash1: input = %d chars, time = %d ms\n",
               i, GetTickCount() - tickcount);

        //Reset the string - this test is destructive.
        for(j = 0; j < i - 5; j += 3)
        {
            input[j] = '\\';
            input[j+1] = '\\';
            input[j+2] = 'Z';
        }

        //Remember that j was incremented before the conditional
        //was checked.
        strncpy(input + j, end, 4);

        tickcount = GetTickCount();
        StripBackslash2(input);
        printf("StripBackslash2: input = %d chars, time = %d ms\n",
               i, GetTickCount() - tickcount);

        free(input);
    }

    return 0;
}
```